第一部分：Python基础语法与核心库

1.1 输入输出与字符串处理

- 标准输入输出技巧
- 字符串操作与格式化
- 正则表达式基础

标准输入输出技巧

基础输入输出

```
# 单行输入
n = int(input())                    # 输入整数
s = input()                         # 输入字符串
s = input().strip()                 # 去除首尾空白字符

# 多个值输入
a, b = input().split()              # 字符串分割
a, b = map(int, input().split())    # 转换为整数
nums = list(map(int, input().split()))  # 整数列表

# 指定分隔符
a, b, c = input().split(',')        # 逗号分隔
```

快速输入输出（大数据量优化）

```
import sys

# 快速输入
input = sys.stdin.readline
# 或者
input = lambda: sys.stdin.readline().strip()

# 快速输出
print = sys.stdout.write

# 示例：快速读取大量数据
def fast_io_example():
    input = sys.stdin.readline
    n = int(input())

    # 读取n行数据
    data = []
    for _ in range(n):
        line = input().strip()
        data.append(line)

    # 批量输出
    sys.stdout.write('\n'.join(results) + '\n')
```

多行输入处理

```
# 方法1：已知行数
n = int(input())
lines = []
for _ in range(n):
    lines.append(input().strip())

# 方法2：读取到EOF
import sys
lines = []
for line in sys.stdin:
    lines.append(line.strip())

# 方法3：使用列表推导
n = int(input())
lines = [input().strip() for _ in range(n)]

# 方法4：读取矩阵
n, m = map(int, input().split())
matrix = []
for _ in range(n):
    row = list(map(int, input().split()))
    matrix.append(row)

# 或者一行写法
matrix = [list(map(int, input().split())) for _ in range(n)]
```

特殊输入格式处理

```
# 处理不定长输入
def read_until_empty():
    lines = []
    while True:
        try:
            line = input().strip()
            if not line:  # 空行结束
                break
            lines.append(line)
        except EOFError:
            break
    return lines

# 处理多组测试用例
def multiple_test_cases():
    while True:
        try:
            line = input().strip()
            if not line:
                break
            # 处理每组数据
            process_case(line)
```

```
        except EOFError:
            break

# 处理以特定值结束的输入
def read_until_zero():
    while True:
        n = int(input())
        if n == 0:
            break
        # 处理数据
        nums = list(map(int, input().split()))
```

输出格式化技巧

```
# 基础输出
print("Hello World")
print(42)
print(3.14159)

# 格式化输出
name = "Alice"
age = 25
score = 95.5

# f-string格式化（推荐）
print(f"姓名: {name}, 年龄: {age}, 分数: {score:.2f}")

# format方法
print("姓名: {}, 年龄: {}, 分数: {:.2f}".format(name, age, score))
print("姓名: {name}, 年龄: {age}, 分数: {score:.2f}".format(
    name=name, age=age, score=score))

# %格式化
print("姓名: %s, 年龄: %d, 分数: %.2f" % (name, age, score))

# 数字格式化
num = 1234567.89
print(f"{num:,.2f}")         # 1,234,567.89 (千分位)
print(f"{num:>15.2f}")       # 右对齐，宽度15
print(f"{num:<15.2f}")       # 左对齐，宽度15
print(f"{num:^15.2f}")       # 居中对齐，宽度15
print(f"{num:015.2f}")       # 用0填充

# 进制转换输出
n = 255
print(f"十进制: {n}")         # 255
print(f"二进制: {n:b}")       # 11111111
print(f"八进制: {n:o}")       # 377
print(f"十六进制: {n:x}")     # ff
print(f"十六进制: {n:X}")     # FF

# 列表输出
```

```python
nums = [1, 2, 3, 4, 5]
print(*nums)                  # 1 2 3 4 5 (空格分隔)
print(*nums, sep=', ')        # 1, 2, 3, 4, 5 (逗号分隔)
print('\n'.join(map(str, nums)))  # 每个数字一行

# 矩阵输出
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
for row in matrix:
    print(*row)

# 或者
for row in matrix:
    print(' '.join(map(str, row)))
```

字符串操作与格式化

字符串基础操作

```python
s = "Hello World Python"

# 长度和索引
print(len(s))               # 18
print(s[0])                 # 'H'
print(s[-1])                # 'n'
print(s[6:11])              # 'World'

# 大小写转换
print(s.upper())            # 'HELLO WORLD PYTHON'
print(s.lower())            # 'hello world python'
print(s.title())            # 'Hello World Python'
print(s.capitalize())       # 'Hello world python'
print(s.swapcase())         # 'hELLO WORLD pYTHON'

# 查找和替换
print(s.find('World'))      # 6 (找到返回索引，没找到返回-1)
print(s.index('World'))     # 6 (找到返回索引，没找到抛异常)
print(s.count('l'))         # 3 (统计出现次数)
print(s.replace('World', 'Python'))  # 'Hello Python Python'

# 字符串判断
print(s.startswith('Hello'))  # True
print(s.endswith('Python'))   # True
print('123'.isdigit())        # True
print('abc'.isalpha())        # True
print('abc123'.isalnum())     # True
print('   '.isspace())        # True
```

字符串分割与连接

```
text = "apple,banana,orange"

# 分割
fruits = text.split(',')      # ['apple', 'banana', 'orange']
words = "hello world".split() # ['hello', 'world'] (默认按空白分割)

# 限制分割次数
data = "a-b-c-d-e"
parts = data.split('-', 2)    # ['a', 'b', 'c-d-e']

# 从右边分割
path = "/home/user/file.txt"
parts = path.rsplit('/', 1)   # ['/home/user', 'file.txt']

# 分割保留分隔符
import re
text = "word1;word2;word3"
parts = re.split('(;)', text) # ['word1', ';', 'word2', ';', 'word3']

# 连接
fruits = ['apple', 'banana', 'orange']
result = ','.join(fruits)     # 'apple,banana,orange'
result = ' | '.join(fruits)   # 'apple | banana | orange'

# 连接数字列表
nums = [1, 2, 3, 4, 5]
result = ','.join(map(str, nums))  # '1,2,3,4,5'
```

字符串去除空白和填充

```
text = "  hello world  "

# 去除空白
print(text.strip())          # 'hello world'
print(text.lstrip())         # 'hello world  '
print(text.rstrip())         # '  hello world'
print(text.strip(' h'))      # 'ello world' (去除指定字符)

# 填充
print("hello".center(10))    # '  hello   '
print("hello".ljust(10))     # 'hello     '
print("hello".rjust(10))     # '     hello'
print("42".zfill(5))         # '00042' (用0填充)

# 自定义填充字符
print("hello".center(10, '*'))  # '**hello***'
```

字符串编码与转换

```
# 字符与ASCII码转换
print(ord('A'))             # 65
print(chr(65))              # 'A'

# 字符串与字节转换
text = "Hello 世界"
encoded = text.encode('utf-8')
decoded = encoded.decode('utf-8')

# 字符串与列表转换
s = "hello"
char_list = list(s)         # ['h', 'e', 'l', 'l', 'o']
s_back = ''.join(char_list) # 'hello'

# 反转字符串
s = "hello"
reversed_s = s[::-1]        # 'olleh'
reversed_s = ''.join(reversed(s))  # 'olleh'
```

高级字符串操作

```
# 字符串模板
from string import Template

template = Template("Hello $name, your score is $score")
result = template.substitute(name="Alice", score=95)

# 安全替换（忽略缺失的变量）
result = template.safe_substitute(name="Alice")

# 字符串翻译
# 创建翻译表
trans = str.maketrans("aeiou", "12345")
text = "hello world"
translated = text.translate(trans)  # 'h2ll4 w4rld'

# 删除字符
delete_vowels = str.maketrans("", "", "aeiou")
no_vowels = text.translate(delete_vowels)  # 'hll wrld'

# 字符串比较
import operator
words = ["apple", "Banana", "cherry"]
words.sort()                          # ['Banana', 'apple', 'cherry']
words.sort(key=str.lower)             # ['apple', 'Banana', 'cherry']
words.sort(key=len)                   # ['apple', 'cherry', 'Banana']

# 自然排序（处理数字）
def natural_sort_key(s):
    import re
```

```
    return [int(text) if text.isdigit() else text.lower()
            for text in re.split('([0-9]+)', s)]

files = ["file10.txt", "file2.txt", "file1.txt"]
files.sort(key=natural_sort_key)     # ['file1.txt', 'file2.txt', 'file10.txt']
```

正则表达式基础

正则表达式导入和基础使用

```
import re

# 基础匹配函数
text = "The price is $123.45 and tax is $12.34"

# re.search() - 查找第一个匹配
match = re.search(r'\$(\d+\.\d+)', text)
if match:
    print(match.group())    # '$123.45'
    print(match.group(1))   # '123.45' (第一个捕获组)

# re.findall() - 查找所有匹配
prices = re.findall(r'\$(\d+\.\d+)', text)
print(prices)               # ['123.45', '12.34']

# re.finditer() - 返回匹配对象的迭代器
for match in re.finditer(r'\$(\d+\.\d+)', text):
    print(f"Found {match.group()} at position {match.start()}-{match.end()}")

# re.match() - 从字符串开始匹配
email = "user@example.com"
match = re.match(r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$', email)
if match:
    print("Valid email")
```

常用正则表达式模式

```
# 基础字符类
patterns = {
    r'\d': '数字',           # [0-9]
    r'\D': '非数字',         # [^0-9]
    r'\w': '字母数字下划线',  # [a-zA-Z0-9_]
    r'\W': '非字母数字下划线', # [^a-zA-Z0-9_]
    r'\s': '空白字符',       # [ \t\n\r\f\v]
    r'\S': '非空白字符',     # [^ \t\n\r\f\v]
    r'.': '任意字符(除换行)',
    r'^': '行首',
    r'$': '行尾',
}

# 量词
quantifiers = {
```

```
    r'*': '0次或多次',
    r'+': '1次或多次',
    r'?': '0次或1次',
    r'{n}': '恰好n次',
    r'{n,}': 'n次或多次',
    r'{n,m}': 'n到m次',
}

# 实用正则表达式
def common_patterns():
    # 邮箱验证
    email_pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'

    # 手机号验证（中国）
    phone_pattern = r'^1[3-9]\d{9}$'

    # 身份证号验证
    id_pattern = r'^\d{17}[\dXx]$'

    # URL验证
    url_pattern = r'^https?://[^\s/$.?#].[^\s]*$'

    # IP地址验证
    ip_pattern = r'^(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$'

    # 中文字符
    chinese_pattern = r'[\u4e00-\u9fa5]'

    # 密码强度（至少8位，包含大小写字母和数字）
    password_pattern = r'^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)[a-zA-Z\d@$!%*?&]{8,}$'

    return {
        'email': email_pattern,
        'phone': phone_pattern,
        'id': id_pattern,
        'url': url_pattern,
        'ip': ip_pattern,
        'chinese': chinese_pattern,
        'password': password_pattern
    }
```

正则表达式高级操作

```
# 编译正则表达式（提高性能）
pattern = re.compile(r'\d+')
numbers = pattern.findall("There are 123 and 456 numbers")

# 分组和命名分组
text = "John Smith, age 25, email: john@example.com"

# 普通分组
```

```
pattern = r'(\w+)\s+(\w+),\s+age\s+(\d+),\s+email:\s+([^,\s]+)'
match = re.search(pattern, text)
if match:
    first_name, last_name, age, email = match.groups()
    print(f"Name: {first_name} {last_name}, Age: {age}, Email: {email}")

# 命名分组
pattern = r'(?P<first>\w+)\s+(?P<last>\w+),\s+age\s+(?P<age>\d+),\s+email:\s+(?P<email>
[^,\s]+)'
match = re.search(pattern, text)
if match:
    print(match.groupdict())
    print(f"First name: {match.group('first')}")

# 替换操作
text = "The price is $123.45"

# 基础替换
new_text = re.sub(r'\$(\d+\.\d+)', r'¥\1', text)  # "The price is ¥123.45"

# 使用函数进行替换
def currency_converter(match):
    price = float(match.group(1))
    return f"¥{price * 6.5:.2f}"

new_text = re.sub(r'\$(\d+\.\d+)', currency_converter, text)

# 分割字符串
text = "apple,banana;orange:grape"
fruits = re.split(r'[,;:]', text)  # ['apple', 'banana', 'orange', 'grape']

# 保留分隔符
fruits_with_sep = re.split(r'([,;:])', text)
```

正则表达式标志

```
# 常用标志
flags = {
    're.IGNORECASE': '忽略大小写',
    're.MULTILINE': '多行模式',
    're.DOTALL': '点号匹配换行符',
    're.VERBOSE': '详细模式（可以写注释）',
}

# 使用示例
text = "Hello\nWorld"

# 忽略大小写
pattern = re.compile(r'hello', re.IGNORECASE)
match = pattern.search(text)  # 可以匹配 "Hello"

# 多行模式
```

```
text = "line1\nline2\nline3"
matches = re.findall(r'^line', text, re.MULTILINE)  # ['line', 'line', 'line']

# 详细模式
pattern = re.compile(r'''
    ^                   # 行首
    (\w+)               # 第一个单词
    \s+                 # 空白字符
    (\w+)               # 第二个单词
    $                   # 行尾
''', re.VERBOSE)

# 组合标志
pattern = re.compile(r'hello.*world', re.IGNORECASE | re.DOTALL)
```

实战应用示例

python

运行复制

```
# 日志解析
def parse_log_file(log_content):
    # 解析Apache日志格式
    log_pattern = r'(\S+) - - \[([^\]]+)\] "(\S+) (\S+) (\S+)" (\d+) (\d+)'

    logs = []
    for line in log_content.split('\n'):
        match = re.match(log_pattern, line)
        if match:
            ip, timestamp, method, url, protocol, status, size = match.groups()
            logs.append({
                'ip': ip,
                'timestamp': timestamp,
                'method': method,
                'url': url,
                'status': int(status),
                'size': int(size) if size != '-' else 0
            })
    return logs

# 数据清洗
def clean_text(text):
    # 移除HTML标签
    text = re.sub(r'<[^>]+>', '', text)

    # 移除多余的空白
    text = re.sub(r'\s+', ' ', text)

    # 移除特殊字符，保留中英文和数字
    text = re.sub(r'[^\w\s\u4e00-\u9fa5]', '', text)

    return text.strip()
```

```python
# 提取信息
def extract_info(text):
    # 提取所有邮箱
    emails = re.findall(r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b', text)

    # 提取所有手机号
    phones = re.findall(r'1[3-9]\d{9}', text)

    # 提取所有URL
    urls = re.findall(r'https?://[^\s]+', text)

    return {
        'emails': emails,
        'phones': phones,
        'urls': urls
    }

# 验证函数
def validate_input(data_type, value):
    patterns = common_patterns()

    if data_type in patterns:
        return bool(re.match(patterns[data_type], value))
    return False

# 使用示例
if __name__ == "__main__":
    # 测试邮箱验证
    email = "test@example.com"
    print(f"Email {email} is valid: {validate_input('email', email)}")

    # 测试文本清洗
    dirty_text = "<p>Hello   世界!</p>  @#$%"
    clean = clean_text(dirty_text)
    print(f"Clean text: {clean}")

    # 提取信息
    sample_text = "Contact us at admin@example.com or call 13812345678. Visit
https://example.com"
    info = extract_info(sample_text)
    print(f"Extracted info: {info}")
```

1.2 基础数据结构操作

- 列表、元组、字典、集合的高效操作
- 列表推导式与生成器
- 切片技巧

列表、元组、字典、集合的高效操作

列表(List)高效操作

python

运行复制

```python
# 创建列表的多种方式
lst1 = [1, 2, 3, 4, 5]
lst2 = list(range(10))              # [0, 1, 2, ..., 9]
lst3 = [0] * 5                      # [0, 0, 0, 0, 0]
lst4 = list("hello")                # ['h', 'e', 'l', 'l', 'o']

# 高效添加元素
lst = []
lst.append(1)                       # 末尾添加，O(1)
lst.extend([2, 3, 4])               # 批量添加，比循环append快
lst.insert(0, 0)                    # 指定位置插入，O(n)

# 高效删除元素
lst = [1, 2, 3, 4, 5, 3, 6]
lst.remove(3)                       # 删除第一个值为3的元素，O(n)
popped = lst.pop()                  # 删除并返回最后一个元素，O(1)
popped = lst.pop(0)                 # 删除并返回指定位置元素，O(n)
del lst[1:3]                        # 删除切片范围内的元素

# 查找操作
lst = [1, 2, 3, 4, 5, 3, 6]
index = lst.index(3)                # 返回第一个值为3的索引
count = lst.count(3)                # 统计值为3的元素个数

# 高效排序和反转
lst = [3, 1, 4, 1, 5, 9, 2, 6]
lst.sort()                          # 原地排序，O(nlogn)
lst.sort(reverse=True)              # 降序排序
lst.sort(key=len)                   # 自定义排序键
sorted_lst = sorted(lst)            # 返回新的排序列表

lst.reverse()                       # 原地反转，O(n)
reversed_lst = lst[::-1]            # 返回新的反转列表

# 列表的数学运算
lst1 = [1, 2, 3]
lst2 = [4, 5, 6]
combined = lst1 + lst2              # [1, 2, 3, 4, 5, 6]
repeated = lst1 * 3                 # [1, 2, 3, 1, 2, 3, 1, 2, 3]

# 列表的比较
lst1 = [1, 2, 3]
lst2 = [1, 2, 4]
print(lst1 < lst2)                  # True (按字典序比较)
```

```python
# 高效的列表操作技巧
def efficient_list_operations():
    # 预分配空间（当知道最终大小时）
    n = 1000
    lst = [None] * n
    for i in range(n):
        lst[i] = i * i

    # 使用deque进行频繁的首部操作
    from collections import deque
    dq = deque([1, 2, 3])
    dq.appendleft(0)                # O(1)
    dq.popleft()                    # O(1)

    # 列表去重（保持顺序）
    def remove_duplicates(lst):
        seen = set()
        result = []
        for item in lst:
            if item not in seen:
                seen.add(item)
                result.append(item)
        return result

    # 或使用dict.fromkeys()
    def remove_duplicates_v2(lst):
        return list(dict.fromkeys(lst))
```

元组(Tuple)高效操作

python

运行复制

```python
# 创建元组
tup1 = (1, 2, 3)
tup2 = 1, 2, 3                      # 括号可选
tup3 = (1,)                         # 单元素元组需要逗号
tup4 = tuple([1, 2, 3])             # 从列表创建

# 元组解包
point = (3, 4)
x, y = point                        # x=3, y=4

# 多重赋值
a, b = b, a                         # 交换变量

# 函数返回多个值
def get_name_age():
    return "Alice", 25

name, age = get_name_age()
```

```python
# 使用*操作符
first, *middle, last = (1, 2, 3, 4, 5)  # first=1, middle=[2,3,4], last=5

# 具名元组 - 更好的可读性
from collections import namedtuple

Point = namedtuple('Point', ['x', 'y'])
p = Point(3, 4)
print(p.x, p.y)                    # 3 4
print(p[0], p[1])                   # 3 4 (仍然可以用索引)

# 具名元组的方法
p2 = p._replace(x=5)                # Point(x=5, y=4)
print(p._asdict())                  # {'x': 3, 'y': 4}

# 元组作为字典键（利用不可变特性）
points = {}
points[(0, 0)] = "origin"
points[(1, 1)] = "diagonal"

# 元组的性能优势
def tuple_vs_list_performance():
    import timeit

    # 创建时间比较
    list_time = timeit.timeit(lambda: [1, 2, 3, 4, 5], number=1000000)
    tuple_time = timeit.timeit(lambda: (1, 2, 3, 4, 5), number=1000000)

    print(f"List creation: {list_time}")
    print(f"Tuple creation: {tuple_time}")  # 通常更快
```

字典(Dictionary)高效操作

python

运行复制

```python
# 创建字典的多种方式
dict1 = {'a': 1, 'b': 2, 'c': 3}
dict2 = dict(a=1, b=2, c=3)
dict3 = dict([('a', 1), ('b', 2), ('c', 3)])
dict4 = {x: x**2 for x in range(5)}  # 字典推导式

# 高效访问和修改
d = {'a': 1, 'b': 2}

# 安全访问
value = d.get('c', 0)               # 不存在返回默认值0
value = d.setdefault('c', 0)        # 不存在则设置并返回默认值

# 批量更新
d.update({'d': 4, 'e': 5})
d.update([('f', 6), ('g', 7)])
```

```python
d.update(h=8, i=9)

# 字典合并（Python 3.9+）
dict1 = {'a': 1, 'b': 2}
dict2 = {'c': 3, 'd': 4}
merged = dict1 | dict2              # {'a': 1, 'b': 2, 'c': 3, 'd': 4}

# 字典解包合并（Python 3.5+）
merged = {**dict1, **dict2}

# 高效遍历
d = {'a': 1, 'b': 2, 'c': 3}

# 遍历键
for key in d:
    print(key)

# 遍历值
for value in d.values():
    print(value)

# 遍历键值对
for key, value in d.items():
    print(key, value)

# 字典的高级操作
from collections import defaultdict, Counter

# defaultdict - 自动创建默认值
dd = defaultdict(list)
dd['key1'].append(1)                # 自动创建空列表

dd_int = defaultdict(int)
dd_int['count'] += 1                # 自动初始化为0

# Counter - 计数字典
text = "hello world"
char_count = Counter(text)
print(char_count.most_common(3))    # [('l', 3), ('o', 2), (' ', 1)]

# 字典的集合操作
dict1 = {'a': 1, 'b': 2, 'c': 3}
dict2 = {'b': 2, 'c': 4, 'd': 5}

# 键的集合操作
common_keys = dict1.keys() & dict2.keys()        # {'b', 'c'}
diff_keys = dict1.keys() - dict2.keys()          # {'a'}
all_keys = dict1.keys() | dict2.keys()           # {'a', 'b', 'c', 'd'}

# 字典排序
sorted_by_key = dict(sorted(dict1.items()))
sorted_by_value = dict(sorted(dict1.items(), key=lambda x: x[1]))
```

```python
# 反转字典
def reverse_dict(d):
    return {v: k for k, v in d.items()}

# 多级字典操作
def deep_get(dictionary, keys, default=None):
    """安全地访问多级字典"""
    for key in keys:
        if isinstance(dictionary, dict) and key in dictionary:
            dictionary = dictionary[key]
        else:
            return default
    return dictionary

nested = {'a': {'b': {'c': 42}}}
value = deep_get(nested, ['a', 'b', 'c'])  # 42
value = deep_get(nested, ['a', 'x', 'y'])  # None
```

集合(Set)高效操作

python

运行复制

```python
# 创建集合
set1 = {1, 2, 3, 4, 5}
set2 = set([1, 2, 3, 3, 4])        # {1, 2, 3, 4} 自动去重
set3 = set("hello")                 # {'h', 'e', 'l', 'o'}
empty_set = set()                   # 空集合，不能用{}

# 集合的基本操作
s = {1, 2, 3}
s.add(4)                            # 添加元素
s.update([5, 6, 7])                 # 批量添加
s.remove(1)                         # 删除元素，不存在会报错
s.discard(1)                        # 删除元素，不存在不报错
popped = s.pop()                    # 随机删除并返回一个元素

# 集合的数学运算
set1 = {1, 2, 3, 4}
set2 = {3, 4, 5, 6}

# 并集
union = set1 | set2                 # {1, 2, 3, 4, 5, 6}
union = set1.union(set2)

# 交集
intersection = set1 & set2          # {3, 4}
intersection = set1.intersection(set2)

# 差集
difference = set1 - set2            # {1, 2}
difference = set1.difference(set2)
```

```python
# 对称差集
sym_diff = set1 ^ set2              # {1, 2, 5, 6}
sym_diff = set1.symmetric_difference(set2)

# 集合的关系判断
set1 = {1, 2, 3}
set2 = {1, 2, 3, 4, 5}

print(set1.issubset(set2))          # True
print(set2.issuperset(set1))        # True
print(set1.isdisjoint({4, 5, 6}))  # True (无交集)

# frozenset - 不可变集合
fs = frozenset([1, 2, 3, 4])
# fs.add(5)  # 错误! frozenset不可变

# 集合可以作为字典的键
cache = {}
key = frozenset(['apple', 'banana'])
cache[key] = "fruit_combination"

# 集合的高效应用
def set_applications():
    # 快速去重
    nums = [1, 2, 2, 3, 3, 4, 5]
    unique = list(set(nums))

    # 快速判断成员资格 O(1)
    valid_ids = {1, 5, 10, 15, 20}
    if user_id in valid_ids:
        print("Valid user")

    # 找出两个列表的公共元素
    list1 = [1, 2, 3, 4, 5]
    list2 = [3, 4, 5, 6, 7]
    common = list(set(list1) & set(list2))

    # 过滤操作
    words = ['apple', 'banana', 'cherry', 'date']
    vowels = set('aeiou')

    # 找出包含元音的单词
    words_with_vowels = [word for word in words
                         if set(word.lower()) & vowels]

    return unique, common, words_with_vowels
```

列表推导式与生成器

列表推导式基础

```
# 基础语法：[expression for item in iterable]
squares = [x**2 for x in range(10)]  # [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

# 带条件的列表推导式
evens = [x for x in range(20) if x % 2 == 0]  # [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

# 复杂表达式
words = ['hello', 'world', 'python']
upper_words = [word.upper() for word in words]  # ['HELLO', 'WORLD', 'PYTHON']

# 嵌套列表推导式
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
flattened = [num for row in matrix for num in row]  # [1, 2, 3, 4, 5, 6, 7, 8, 9]

# 等价的嵌套循环
flattened = []
for row in matrix:
    for num in row:
        flattened.append(num)

# 条件表达式在列表推导式中
numbers = [1, -2, 3, -4, 5]
abs_numbers = [x if x >= 0 else -x for x in numbers]  # [1, 2, 3, 4, 5]

# 多重条件
filtered = [x for x in range(100) if x % 2 == 0 if x % 3 == 0]  # 能被2和3整除的数
```

字典和集合推导式

python

运行复制

```
# 字典推导式
squares_dict = {x: x**2 for x in range(5)}  # {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

# 从两个列表创建字典
keys = ['a', 'b', 'c']
values = [1, 2, 3]
mapping = {k: v for k, v in zip(keys, values)}  # {'a': 1, 'b': 2, 'c': 3}

# 字典过滤和转换
original = {'a': 1, 'b': 2, 'c': 3, 'd': 4}
filtered_dict = {k: v for k, v in original.items() if v % 2 == 0}  # {'b': 2, 'd': 4}

# 反转字典
reversed_dict = {v: k for k, v in original.items()}
```

```
# 集合推导式
vowels_in_words = {char for word in ['hello', 'world'] for char in word if char in 'aeiou'}
```

生成器表达式

```
# 生成器表达式语法：(expression for item in iterable)
squares_gen = (x**2 for x in range(10))  # 生成器对象

# 生成器是惰性求值的
print(type(squares_gen))           # <class 'generator'>

# 遍历生成器
for square in squares_gen:
    print(square)

# 生成器只能遍历一次
squares_gen = (x**2 for x in range(5))
list1 = list(squares_gen)          # [0, 1, 4, 9, 16]
list2 = list(squares_gen)          # [] 空列表，生成器已耗尽

# 内存效率对比
import sys

# 列表推导式 - 立即创建所有元素
list_comp = [x**2 for x in range(1000)]
print(sys.getsizeof(list_comp))    # 较大的内存占用

# 生成器表达式 - 按需生成
gen_exp = (x**2 for x in range(1000))
print(sys.getsizeof(gen_exp))      # 很小的内存占用

# 生成器的实际应用
def process_large_file(filename):
    # 逐行处理大文件，内存效率高
    with open(filename) as f:
        processed_lines = (line.strip().upper() for line in f if line.strip())
        for line in processed_lines:
            yield line  # 进一步处理

# 无限生成器
def fibonacci_generator():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

# 使用无限生成器
fib = fibonacci_generator()
first_10_fibs = [next(fib) for _ in range(10)]
```

自定义生成器函数

```python
# 使用yield关键字创建生成器函数
def countdown(n):
    while n > 0:
        yield n
        n -= 1

# 使用生成器
for num in countdown(5):
    print(num)  # 5, 4, 3, 2, 1

# 生成器的状态保持
def stateful_generator():
    count = 0
    while True:
        value = yield count
        if value is not None:
            count = value
        else:
            count += 1

gen = stateful_generator()
next(gen)          # 0
next(gen)          # 1
gen.send(10)       # 设置count为10，返回10
next(gen)          # 11

# 生成器的实用示例
def batch_generator(data, batch_size):
    """将数据分批处理"""
    for i in range(0, len(data), batch_size):
        yield data[i:i + batch_size]

data = list(range(100))
for batch in batch_generator(data, 10):
    print(f"Processing batch of size {len(batch)}")

def sliding_window(sequence, window_size):
    """滑动窗口生成器"""
    for i in range(len(sequence) - window_size + 1):
        yield sequence[i:i + window_size]

text = "hello world"
for window in sliding_window(text, 3):
    print(window)  # 'hel', 'ell', 'llo', ...
```

切片技巧

基础切片操作

```
# 基础语法：sequence[start:stop:step]
lst = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# 基础切片
print(lst[2:7])        # [2, 3, 4, 5, 6]
print(lst[:5])         # [0, 1, 2, 3, 4] 从开始到索引5
print(lst[5:])         # [5, 6, 7, 8, 9] 从索引5到结束
print(lst[:])          # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] 完整拷贝

# 负索引切片
print(lst[-3:])        # [7, 8, 9] 最后三个元素
print(lst[:-3])        # [0, 1, 2, 3, 4, 5, 6] 除了最后三个
print(lst[-5:-2])      # [5, 6, 7] 负索引范围

# 步长切片
print(lst[::2])        # [0, 2, 4, 6, 8] 每隔一个元素
print(lst[1::2])       # [1, 3, 5, 7, 9] 从索引1开始，每隔一个
print(lst[::3])        # [0, 3, 6, 9] 每隔两个元素
```

高级切片技巧

```
# 反转序列
lst = [1, 2, 3, 4, 5]
reversed_lst = lst[::-1]           # [5, 4, 3, 2, 1]

# 字符串反转
text = "hello"
reversed_text = text[::-1]         # "olleh"

# 部分反转
lst = [1, 2, 3, 4, 5, 6]
lst[1:4] = lst[1:4][::-1]         # [1, 4, 3, 2, 5, 6]

# 删除操作
lst = [1, 2, 3, 4, 5, 6, 7, 8, 9]
del lst[2:5]                       # [1, 2, 6, 7, 8, 9]
del lst[::2]                       # 删除偶数位置的元素

# 替换操作
lst = [1, 2, 3, 4, 5]
lst[1:4] = [20, 30]                # [1, 20, 30, 5] 长度可以不同

# 插入操作
lst = [1, 2, 3, 4, 5]
lst[2:2] = [2.5]                   # [1, 2, 2.5, 3, 4, 5] 在索引2处插入

# 多维列表切片
matrix = [
```

```
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

# 获取第一列
first_column = [row[0] for row in matrix]  # [1, 4, 7]

# 获取子矩阵
sub_matrix = [row[1:] for row in matrix[1:]]  # [[5, 6], [8, 9]]

# 使用NumPy风格的切片（需要自定义实现）
def get_column(matrix, col_index):
    return [row[col_index] for row in matrix]

def get_submatrix(matrix, row_slice, col_slice):
    return [row[col_slice] for row in matrix[row_slice]]
```

切片的性能考虑

```
import timeit

# 切片 vs 循环的性能比较
def slice_copy(lst):
    return lst[:]

def loop_copy(lst):
    result = []
    for item in lst:
        result.append(item)
    return result

# 大列表性能测试
large_list = list(range(10000))

slice_time = timeit.timeit(lambda: slice_copy(large_list), number=1000)
loop_time = timeit.timeit(lambda: loop_copy(large_list), number=1000)

print(f"Slice copy: {slice_time:.4f} seconds")
print(f"Loop copy: {loop_time:.4f} seconds")  # 切片通常更快

# 浅拷贝 vs 深拷贝
import copy

nested_list = [[1, 2], [3, 4], [5, 6]]

# 浅拷贝
shallow = nested_list[:]           # 或 nested_list.copy()
shallow[0][0] = 999                # 影响原列表

# 深拷贝
deep = copy.deepcopy(nested_list)
```

```
deep[0][0] = 999                    # 不影响原列表
```

切片的实用应用

```
# 文本处理中的切片应用
def text_processing_examples():
    text = "Hello, World! This is Python."

    # 提取文件扩展名
    filename = "document.pdf"
    extension = filename[filename.rfind('.')+1:]  # "pdf"

    # 更安全的方式
    if '.' in filename:
        extension = filename.split('.')[-1]

    # 移除字符串前缀和后缀
    def remove_prefix(text, prefix):
        if text.startswith(prefix):
            return text[len(prefix):]
        return text

    def remove_suffix(text, suffix):
        if text.endswith(suffix):
            return text[:-len(suffix)]
        return text

    # 字符串中间省略
    def truncate_middle(text, max_length):
        if len(text) <= max_length:
            return text

        if max_length <= 3:
            return text[:max_length]

        # 计算两端保留的字符数
        side_length = (max_length - 3) // 2
        return text[:side_length] + "..." + text[-side_length:]

    # 示例
    long_text = "This is a very long string that needs to be truncated"
    short = truncate_middle(long_text, 20)  # "This is...truncated"

    return extension, short

# 数据分析中的切片应用
def data_analysis_slicing():
    # 时间序列数据处理
    dates = ['2023-01', '2023-02', '2023-03', '2023-04', '2023-05', '2023-06']
    values = [100, 120, 110, 130, 125, 140]

    # 获取最近3个月的数据
```

```
    recent_dates = dates[-3:]
    recent_values = values[-3:]

    # 移动平均窗口
    def moving_average(data, window_size):
        return [sum(data[i:i+window_size]) / window_size
                for i in range(len(data) - window_size + 1)]

    ma_3 = moving_average(values, 3)

    # 数据分页
    def paginate(data, page_size):
        return [data[i:i+page_size] for i in range(0, len(data), page_size)]

    pages = paginate(values, 2)  # [[100, 120], [110, 130], [125, 140]]

    return recent_dates, recent_values, ma_3, pages

# 算法中的切片技巧
def algorithm_slicing_tricks():
    # 快速排序中的分区
    def quicksort(arr):
        if len(arr) <= 1:
            return arr

        pivot = arr[len(arr) // 2]
        left = [x for x in arr if x < pivot]
        middle = [x for x in arr if x == pivot]
        right = [x for x in arr if x > pivot]

        return quicksort(left) + middle + quicksort(right)

    # 滑动窗口最大值
    def sliding_window_max(nums, k):
        if not nums or k == 0:
            return []

        result = []
        for i in range(len(nums) - k + 1):
            window = nums[i:i+k]
            result.append(max(window))

        return result

    # 数组旋转
    def rotate_array(nums, k):
        n = len(nums)
        k = k % n
        return nums[-k:] + nums[:-k]

    # 示例使用
    arr = [3, 6, 2, 8, 1, 9, 4]
    sorted_arr = quicksort(arr)
```

```
    window_max = sliding_window_max([1, 3, -1, -3, 5, 3, 6, 7], 3)

    rotated = rotate_array([1, 2, 3, 4, 5, 6, 7], 3)  # [5, 6, 7, 1, 2, 3, 4]

    return sorted_arr, window_max, rotated

# 内存高效的切片操作
def memory_efficient_slicing():
    # 使用itertools.islice进行内存高效的切片
    import itertools

    # 对于大型可迭代对象，避免创建中间列表
    def process_large_data(data_generator, start, stop):
        sliced_data = itertools.islice(data_generator, start, stop)
        return list(sliced_data)

    # 生成器切片
    def generator_slice(generator, start, stop, step=1):
        return itertools.islice(generator, start, stop, step)

    # 文件的切片读取
    def read_file_slice(filename, start_line, end_line):
        with open(filename, 'r') as f:
            for i, line in enumerate(f):
                if start_line <= i < end_line:
                    yield line.strip()
                elif i >= end_line:
                    break

    return process_large_data, generator_slice, read_file_slice
```

1.3 核心库导入与使用

- collections模块详解
- itertools迭代工具
- functools函数工具
- math与operator模块

collections模块详解

```
from collections import deque, defaultdict, Counter, OrderedDict, namedtuple

# deque - 双端队列，O(1)的两端操作
dq = deque([1, 2, 3])
dq.appendleft(0)  # 左端添加
dq.popleft()      # 左端删除

# defaultdict - 自动默认值
dd = defaultdict(list)
dd['key'].append(1)  # 自动创建空列表
```

```python
# Counter - 计数器
count = Counter("hello")  # Counter({'l': 2, 'h': 1, 'e': 1, 'o': 1})
count.most_common(2)      # [('l', 2), ('h', 1)]

# namedtuple - 具名元组
Point = namedtuple('Point', ['x', 'y'])
p = Point(1, 2)
print(p.x, p.y)  # 1 2
```

itertools迭代工具

```python
import itertools

# 常用组合工具
list(itertools.permutations([1,2,3], 2))      # 排列
list(itertools.combinations([1,2,3,4], 2))    # 组合
list(itertools.product([1,2], [3,4]))         # 笛卡尔积

# 无限迭代器
itertools.count(1, 2)     # 1, 3, 5, 7, ...
itertools.cycle([1,2,3])  # 1, 2, 3, 1, 2, 3, ...
itertools.repeat('A', 3)  # 'A', 'A', 'A'

# 实用工具
list(itertools.chain([1,2], [3,4]))           # [1, 2, 3, 4] 连接
list(itertools.islice(range(100), 5, 10))     # [5, 6, 7, 8, 9] 切片
```

functools函数工具

```python
from functools import lru_cache, reduce, partial

# lru_cache - 记忆化缓存
@lru_cache(maxsize=None)
def fibonacci(n):
    if n <= 1: return n
    return fibonacci(n-1) + fibonacci(n-2)

# reduce - 累积操作
reduce(lambda x, y: x+y, [1,2,3,4,5])  # 15 (求和)

# partial - 偏函数
def multiply(x, y): return x * y
double = partial(multiply, 2)
print(double(5))  # 10
```

math与operator模块

```
import math
import operator

# math常用函数
math.gcd(12, 18)       # 6 最大公约数
math.sqrt(16)          # 4.0 平方根
math.factorial(5)      # 120 阶乘
math.ceil(3.2)         # 4 向上取整
math.floor(3.8)        # 3 向下取整

# operator - 操作符函数版本
operator.add(1, 2)     # 3 等同于 1 + 2
operator.mul(3, 4)     # 12 等同于 3 * 4
operator.itemgetter(1)([1,2,3])  # 2 获取索引1的元素

# 在排序中使用
data = [('Alice', 25), ('Bob', 30), ('Charlie', 20)]
sorted(data, key=operator.itemgetter(1))  # 按年龄排序
```

快速记忆要点

```
# 考试中最常用的快速导入
from collections import deque, defaultdict, Counter
import heapq, bisect, itertools, math
from functools import lru_cache

# 一行解决常见问题
dd = defaultdict(int)      # 计数器
dq = deque()               # 队列操作
Counter(text).most_common(1)[0][0]  # 最频繁字符
```

第二部分：算法基础

2.1 排序与搜索算法

- 内置排序函数与自定义排序
- 二分查找及其变种
- 快速排序、归并排序实现

内置排序函数与自定义排序

```
# 基础排序
nums = [3, 1, 4, 1, 5, 9, 2, 6]
nums.sort()                    # 原地排序 [1, 1, 2, 3, 4, 5, 6, 9]
sorted_nums = sorted(nums)     # 返回新列表

# 自定义排序
```

```python
data = [('Alice', 25), ('Bob', 30), ('Charlie', 20)]
data.sort(key=lambda x: x[1])          # 按年龄排序
data.sort(key=lambda x: x[0])          # 按姓名排序
data.sort(key=lambda x: (x[1], x[0]))  # 先年龄后姓名

# 常用排序技巧
words = ['apple', 'pie', 'Washington', 'book']
words.sort(key=len)                    # 按长度排序
words.sort(key=str.lower)              # 忽略大小写
words.sort(reverse=True)               # 降序

# 多级排序
students = [('Alice', 85, 'A'), ('Bob', 90, 'B'), ('Charlie', 85, 'A')]
students.sort(key=lambda x: (-x[1], x[0]))  # 分数降序，姓名升序
```

二分查找及其变种

```python
import bisect

# 标准库二分查找
nums = [1, 3, 4, 7, 9, 12, 15]
pos = bisect.bisect_left(nums, 7)   # 3 (插入位置)
pos = bisect.bisect_right(nums, 7)  # 4

# 手写二分查找模板
def binary_search(nums, target):
    left, right = 0, len(nums) - 1
    while left <= right:
        mid = (left + right) // 2
        if nums[mid] == target:
            return mid
        elif nums[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1

# 查找第一个/最后一个位置
def find_first(nums, target):
    left, right = 0, len(nums) - 1
    result = -1
    while left <= right:
        mid = (left + right) // 2
        if nums[mid] == target:
            result = mid
            right = mid - 1  # 继续向左找
        elif nums[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return result
```

```python
def find_last(nums, target):
    left, right = 0, len(nums) - 1
    result = -1
    while left <= right:
        mid = (left + right) // 2
        if nums[mid] == target:
            result = mid
            left = mid + 1   # 继续向右找
        elif nums[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return result

# 二分答案模板
def check(x):
    # 检查x是否满足条件
    pass

def binary_answer(left, right):
    while left < right:
        mid = (left + right) // 2
        if check(mid):
            right = mid
        else:
            left = mid + 1
    return left
```

快速排序、归并排序实现

```python
# 快速排序
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quicksort(left) + middle + quicksort(right)

# 原地快速排序
def quicksort_inplace(arr, low=0, high=None):
    if high is None:
        high = len(arr) - 1
    if low < high:
        pi = partition(arr, low, high)
        quicksort_inplace(arr, low, pi - 1)
        quicksort_inplace(arr, pi + 1, high)

def partition(arr, low, high):
    pivot = arr[high]
    i = low - 1
```

```
    for j in range(low, high):
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1

# 归并排序
def mergesort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = mergesort(arr[:mid])
    right = mergesort(arr[mid:])
    return merge(left, right)

def merge(left, right):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result

# 考试常用：计数排序（适用于范围小的整数）
def counting_sort(arr, max_val):
    count = [0] * (max_val + 1)
    for num in arr:
        count[num] += 1

    result = []
    for i, cnt in enumerate(count):
        result.extend([i] * cnt)
    return result
```

实战技巧总结

```
# 排序算法选择
# - 小数据量：插入排序
# - 一般情况：内置sort()
# - 需要稳定：归并排序
# - 范围小整数：计数排序

# 二分查找使用场景
# - 有序数组查找
# - 查找插入位置
```

```
# - 二分答案（最大值最小化问题）

# 时间复杂度记忆
# - 排序：O(nlogn) 比较排序下界
# - 二分查找：O(logn)
# - 计数排序：O(n+k) k为数据范围
```

2.2 递归与分治

- 递归基础与优化
- 记忆化搜索
- 分治算法经典问题

递归基础与优化

递归基础模板

```python
def recursive_function(n):
    # 1. 基础情况（递归终止条件）
    if n <= 1:
        return 1

    # 2. 递归情况（问题分解）
    return recursive_function(n-1) + recursive_function(n-2)

# 经典递归问题
def factorial(n):
    if n <= 1:
        return 1
    return n * factorial(n-1)

def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

# 汉诺塔问题
def hanoi(n, source, target, auxiliary):
    if n == 1:
        print(f"Move disk from {source} to {target}")
        return
    hanoi(n-1, source, auxiliary, target)
    print(f"Move disk from {source} to {target}")
    hanoi(n-1, auxiliary, target, source)
```

递归优化技巧

```python
import sys
sys.setrecursionlimit(10000)  # 增加递归深度限制

# 1. 尾递归优化（Python不直接支持，但可以改写为迭代）
```

```python
def factorial_tail(n, acc=1):
    if n <= 1:
        return acc
    return factorial_tail(n-1, n*acc)

# 改写为迭代版本
def factorial_iterative(n):
    result = 1
    for i in range(1, n+1):
        result *= i
    return result

# 2. 避免重复计算 - 使用参数传递
def fibonacci_optimized(n, a=0, b=1):
    if n == 0:
        return a
    return fibonacci_optimized(n-1, b, a+b)

# 3. 递归转迭代（使用栈模拟）
def factorial_stack(n):
    stack = []
    stack.append(n)
    result = 1

    while stack:
        current = stack.pop()
        if current <= 1:
            continue
        result *= current
        if current > 1:
            stack.append(current - 1)

    return result
```

记忆化搜索

基础记忆化

```python
# 方法1：手动缓存
def fibonacci_memo():
    cache = {}
    def fib(n):
        if n in cache:
            return cache[n]
        if n <= 1:
            result = n
        else:
            result = fib(n-1) + fib(n-2)
        cache[n] = result
        return result
    return fib
```

```python
# 方法2：装饰器缓存
from functools import lru_cache

@lru_cache(maxsize=None)
def fibonacci_lru(n):
    if n <= 1:
        return n
    return fibonacci_lru(n-1) + fibonacci_lru(n-2)

# 方法3：自定义记忆化装饰器
def memoize(func):
    cache = {}
    def wrapper(*args):
        if args in cache:
            return cache[args]
        result = func(*args)
        cache[args] = result
        return result
    return wrapper

@memoize
def fibonacci_custom(n):
    if n <= 1:
        return n
    return fibonacci_custom(n-1) + fibonacci_custom(n-2)
```

记忆化搜索经典应用

```python
# 1. 最长递增子序列(LIS)
@lru_cache(maxsize=None)
def lis_memo(nums, i, prev):
    if i == len(nums):
        return 0

    # 不选择当前元素
    skip = lis_memo(nums, i+1, prev)

    # 选择当前元素（如果可以）
    take = 0
    if prev == -1 or nums[i] > nums[prev]:
        take = 1 + lis_memo(nums, i+1, i)

    return max(skip, take)

# 2. 编辑距离
@lru_cache(maxsize=None)
def edit_distance(s1, s2, i=0, j=0):
    if i == len(s1):
        return len(s2) - j
    if j == len(s2):
        return len(s1) - i
```

```
    if s1[i] == s2[j]:
        return edit_distance(s1, s2, i+1, j+1)

    # 三种操作：插入、删除、替换
    insert = 1 + edit_distance(s1, s2, i, j+1)
    delete = 1 + edit_distance(s1, s2, i+1, j)
    replace = 1 + edit_distance(s1, s2, i+1, j+1)

    return min(insert, delete, replace)

# 3. 路径计数问题
@lru_cache(maxsize=None)
def count_paths(m, n, x=0, y=0):
    if x == m-1 and y == n-1:
        return 1
    if x >= m or y >= n:
        return 0

    return count_paths(m, n, x+1, y) + count_paths(m, n, x, y+1)

# 4. 背包问题
@lru_cache(maxsize=None)
def knapsack_memo(weights, values, capacity, i=0):
    if i == len(weights) or capacity == 0:
        return 0

    # 不选择当前物品
    skip = knapsack_memo(weights, values, capacity, i+1)

    # 选择当前物品（如果容量够）
    take = 0
    if weights[i] <= capacity:
        take = values[i] + knapsack_memo(weights, values, capacity-weights[i], i+1)

    return max(skip, take)
```

分治算法经典问题

分治算法基本模板

```
def divide_and_conquer(problem):
    # 1. 基础情况
    if is_base_case(problem):
        return solve_base_case(problem)

    # 2. 分解问题
    subproblems = divide(problem)

    # 3. 递归解决子问题
    subresults = []
    for subproblem in subproblems:
        subresults.append(divide_and_conquer(subproblem))
```

```
    # 4. 合并结果
    return combine(subresults)
```

经典分治问题

1. 归并排序

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    # 分解
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])

    # 合并
    return merge(left, right)

def merge(left, right):
    result = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    result.extend(left[i:])
    result.extend(right[j:])
    return result
```

2. 快速幂算法

```
def power(base, exp, mod=None):
    if exp == 0:
        return 1
    if exp == 1:
        return base if mod is None else base % mod

    # 分治：a^n = (a^(n/2))^2 或 a * (a^(n/2))^2
    half = power(base, exp // 2, mod)
    half_squared = half * half
    if mod:
        half_squared %= mod

    if exp % 2 == 0:
        return half_squared
```

```
    else:
        result = base * half_squared
        return result if mod is None else result % mod

# 迭代版本（更高效）
def power_iterative(base, exp, mod=None):
    result = 1
    base = base if mod is None else base % mod

    while exp > 0:
        if exp % 2 == 1:
            result = result * base
            if mod:
                result %= mod
        exp //= 2
        base = base * base
        if mod:
            base %= mod

    return result
```

3. 最大子数组和（分治法）

```
def max_subarray_sum_divide(arr):
    def max_crossing_sum(arr, left, mid, right):
        # 包含mid的最大子数组和
        left_sum = float('-inf')
        curr_sum = 0
        for i in range(mid, left-1, -1):
            curr_sum += arr[i]
            left_sum = max(left_sum, curr_sum)

        right_sum = float('-inf')
        curr_sum = 0
        for i in range(mid+1, right+1):
            curr_sum += arr[i]
            right_sum = max(right_sum, curr_sum)

        return left_sum + right_sum

    def max_subarray_helper(arr, left, right):
        if left == right:
            return arr[left]

        mid = (left + right) // 2

        # 三种情况：左半部分、右半部分、跨越中点
        left_sum = max_subarray_helper(arr, left, mid)
        right_sum = max_subarray_helper(arr, mid+1, right)
        cross_sum = max_crossing_sum(arr, left, mid, right)

        return max(left_sum, right_sum, cross_sum)
```

```
    return max_subarray_helper(arr, 0, len(arr)-1)
```

4. 大整数乘法（Karatsuba算法）

```
def karatsuba_multiply(x, y):
    # 基础情况
    if x < 10 or y < 10:
        return x * y

    # 计算位数
    n = max(len(str(x)), len(str(y)))
    half = n // 2

    # 分解
    high1, low1 = divmod(x, 10**half)
    high2, low2 = divmod(y, 10**half)

    # 递归计算
    z0 = karatsuba_multiply(low1, low2)
    z1 = karatsuba_multiply(low1 + high1, low2 + high2)
    z2 = karatsuba_multiply(high1, high2)

    # 合并结果
    return z2 * (10**(2*half)) + (z1 - z2 - z0) * (10**half) + z0
```

5. 寻找最近点对

```
import math

def closest_pair(points):
    def distance(p1, p2):
        return math.sqrt((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2)

    def closest_pair_rec(px, py):
        n = len(px)

        # 基础情况：暴力求解
        if n <= 3:
            min_dist = float('inf')
            for i in range(n):
                for j in range(i+1, n):
                    min_dist = min(min_dist, distance(px[i], px[j]))
            return min_dist

        # 分解
        mid = n // 2
        midpoint = px[mid]

        pyl = [point for point in py if point[0] <= midpoint[0]]
        pyr = [point for point in py if point[0] > midpoint[0]]

        # 递归求解
```

```
        dl = closest_pair_rec(px[:mid], pyl)
        dr = closest_pair_rec(px[mid:], pyr)

        # 找到最小距离
        d = min(dl, dr)

        # 检查跨越中线的点对
        strip = [point for point in py if abs(point[0] - midpoint[0]) < d]

        for i in range(len(strip)):
            j = i + 1
            while j < len(strip) and (strip[j][1] - strip[i][1]) < d:
                d = min(d, distance(strip[i], strip[j]))
                j += 1

        return d

    # 预处理：按x和y坐标排序
    px = sorted(points, key=lambda p: p[0])
    py = sorted(points, key=lambda p: p[1])

    return closest_pair_rec(px, py)
```

2.3 贪心算法

- 贪心策略设计
- 区间调度问题
- 最小生成树问题

贪心策略设计

贪心算法基本思路

python

运行复制

```
# 贪心算法模板
def greedy_algorithm(data):
    # 1. 排序/选择合适的贪心策略
    data.sort(key=greedy_key)

    result = []
    for item in data:
        # 2. 局部最优选择
        if is_feasible(item, result):
            result.append(item)

    return result

# 经典贪心问题：找零钱
def coin_change_greedy(coins, amount):
```

```
    coins.sort(reverse=True)  # 从大到小
    result = []

    for coin in coins:
        while amount >= coin:
            result.append(coin)
            amount -= coin

    return result if amount == 0 else None

# 活动选择问题
def activity_selection(activities):
    # activities: [(start, end), ...]
    # 按结束时间排序
    activities.sort(key=lambda x: x[1])

    selected = [activities[0]]
    last_end = activities[0][1]

    for start, end in activities[1:]:
        if start >= last_end:  # 不冲突
            selected.append((start, end))
            last_end = end

    return selected
```

区间调度问题

区间调度经典问题

python

运行复制

```
# 1. 会议室安排
def meeting_rooms(intervals):
    if not intervals:
        return 0

    intervals.sort(key=lambda x: x[1])  # 按结束时间排序
    count = 1
    end = intervals[0][1]

    for start, finish in intervals[1:]:
        if start >= end:
            count += 1
            end = finish

    return count

# 2. 最少会议室数量
import heapq
```

```python
def min_meeting_rooms(intervals):
    if not intervals:
        return 0

    intervals.sort()  # 按开始时间排序
    heap = []  # 存储结束时间

    for start, end in intervals:
        if heap and heap[0] <= start:
            heapq.heappop(heap)  # 释放会议室
        heapq.heappush(heap, end)  # 分配会议室

    return len(heap)

# 3. 区间覆盖问题
def interval_cover(intervals, target):
    intervals.sort()  # 按起始位置排序
    result = []
    i = 0
    start = target[0]

    while start < target[1] and i < len(intervals):
        if intervals[i][0] > start:
            return None  # 无法覆盖

        # 找到能覆盖start的最长区间
        max_end = 0
        while i < len(intervals) and intervals[i][0] <= start:
            max_end = max(max_end, intervals[i][1])
            i += 1

        result.append((start, max_end))
        start = max_end

    return result if start >= target[1] else None
```

最小生成树问题

Kruskal算法（并查集）

python

运行复制

```python
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]
```

```
    def union(self, x, y):
        px, py = self.find(x), self.find(y)
        if px == py:
            return False
        if self.rank[px] < self.rank[py]:
            px, py = py, px
        self.parent[py] = px
        if self.rank[px] == self.rank[py]:
            self.rank[px] += 1
        return True

def kruskal_mst(n, edges):
    # edges: [(weight, u, v), ...]
    edges.sort()  # 按权重排序
    uf = UnionFind(n)
    mst = []
    total_weight = 0

    for weight, u, v in edges:
        if uf.union(u, v):
            mst.append((u, v, weight))
            total_weight += weight
            if len(mst) == n - 1:
                break

    return mst, total_weight
```

Prim算法（堆优化）

python

运行复制

```
import heapq

def prim_mst(graph, start=0):
    # graph: {node: [(neighbor, weight), ...]}
    n = len(graph)
    visited = [False] * n
    mst = []
    total_weight = 0

    # 优先队列：(weight, from_node, to_node)
    heap = [(0, start, start)]

    while heap and len(mst) < n - 1:
        weight, from_node, to_node = heapq.heappop(heap)

        if visited[to_node]:
            continue

        visited[to_node] = True
```

```
            if from_node != to_node:
                mst.append((from_node, to_node, weight))
                total_weight += weight

            # 添加新的边
            for neighbor, edge_weight in graph[to_node]:
                if not visited[neighbor]:
                    heapq.heappush(heap, (edge_weight, to_node, neighbor))

    return mst, total_weight
```

常见贪心问题总结

python

运行复制

```
# 1. 分发糖果
def candy_distribution(ratings):
    n = len(ratings)
    candies = [1] * n

    # 从左到右
    for i in range(1, n):
        if ratings[i] > ratings[i-1]:
            candies[i] = candies[i-1] + 1

    # 从右到左
    for i in range(n-2, -1, -1):
        if ratings[i] > ratings[i+1]:
            candies[i] = max(candies[i], candies[i+1] + 1)

    return sum(candies)

# 2. 跳跃游戏
def can_jump(nums):
    max_reach = 0
    for i, jump in enumerate(nums):
        if i > max_reach:
            return False
        max_reach = max(max_reach, i + jump)
    return True

# 3. 买卖股票最佳时机
def max_profit(prices):
    profit = 0
    for i in range(1, len(prices)):
        if prices[i] > prices[i-1]:
            profit += prices[i] - prices[i-1]
    return profit

# 4. 加油站
def can_complete_circuit(gas, cost):
```

```
    total_tank = current_tank = start = 0

    for i in range(len(gas)):
        total_tank += gas[i] - cost[i]
        current_tank += gas[i] - cost[i]

        if current_tank < 0:
            start = i + 1
            current_tank = 0

    return start if total_tank >= 0 else -1
```

例子

1.

```
# easy
# 1.两数之和
# 给定一个整数数组 nums 和一个整数目标值 target，请你在该数组中找出 和为目标值 target  的那 两个 整数，并返回它们的数组下标。
# 你可以假设每种输入只会对应一个答案，并且你不能使用两次相同的元素。
# 你可以按任意顺序返回答案。
# 示例 1：
# 输入：nums = [2,7,11,15], target = 9
# 输出：[0,1]
# 解释：因为 nums[0] + nums[1] == 9 ，返回 [0, 1] 。
# 示例 2：
# 输入：nums = [3,2,4], target = 6
# 输出：[1,2]
# 示例 3：
# 输入：nums = [3,3], target = 6
# 输出：[0,1]
# 提示：
# 2 <= nums.length <= 104
# -109 <= nums[i] <= 109
# -109 <= target <= 109
# 只会存在一个有效答案
# 进阶：你可以想出一个时间复杂度小于 O(n^2) 的算法吗？

def twoSum(nums,target):
    hash_map = {}
    for i,num in enumerate(nums):
        tmp = target - num
        if tmp in hash_map:
            return[hash_map[tmp],i]
        hash_map[num]=i
    return []

def main():
    nums_input = input().strip()
```

```
    nums = list(map(int,nums_input.split()))
    target = int(input().strip())
    result = twoSum(nums,target)
    print(result)

if __name__ == "__main__":
    main()
```

2.

```
# medium
# 2.字母异位词分组
# 给你一个字符串数组，请你将 字母异位词 组合在一起。可以按任意顺序返回结果列表。
# 示例 1:
# 输入: strs = ["eat", "tea", "tan", "ate", "nat", "bat"]
# 输出: [["bat"],["nat","tan"],["ate","eat","tea"]]
# 示例 2:
# 输入: strs = [""]
# 输出: [[""]]
# 示例 3:
# 输入: strs = ["a"]
# 输出: [["a"]]
# 提示：
# 1 <= strs.length <= 104
# 0 <= strs[i].length <= 100
# strs[i] 仅包含小写字母

def solution2(strs:list[str])->list[list[str]]:
    alphanum = [2,3,5,7,11,
                13,17,19,23,29,
                31,37,41,43,47,
                53,59,61,67,71,
                73,79,83,89,97,
                101]
    result = dict()
    for str in strs:
        prod = 1
        for item in str:
            prod = alphanum[ord(item)-ord('a')]*prod
        if prod not in result:
            result[prod] = []
        result[prod].append(str)
    return list(result.values())

def main():
    strs = input().strip().split()
    result = solution2(strs)
    print(result)

if __name__ == "__main__":
    main()
```

3.

```python
# medium
# 3. 最长连续序列
# 给定一个未排序的整数数组 nums ，找出数字连续的最长序列（不要求序列元素在原数组中连续）的长度。请你设计并实现时间复杂度为 O(n) 的算法解决此问题。
# 示例 1：
# 输入：nums = [100,4,200,1,3,2]
# 输出：4
# 示例 2：
# 输入：nums = [0,3,7,2,5,8,4,6,0,1]
# 输出：9
# 示例 3：
# 输入：nums = [1,0,1,2]
# 输出：3
# 提示：
# 0 <= nums.length <= 105
# -109 <= nums[i] <= 109

def solution3(nums):
    ans = 0
    set_num = set(nums)
    for x in set_num:
        if x-1 in set_num:
            continue
        y = x + 1
        while y in set_num:
            y += 1
        ans = max(ans,y-x)
    return ans
def main():
    nums_input = input().strip().split()
    nums = list(map(int,nums_input))
    result = solution3(nums)
    print(result)
if __name__ == "__main__":
    main()
```

4.

```python
# 4. 移动零
# 给定一个数组 nums，编写一个函数将所有 0 移动到数组的末尾，同时保持非零元素的相对顺序。请注意 ，必须在不复制数组的情况下原地对数组进行操作。

# 示例 1:
# 输入: nums = [0,1,0,3,12]
# 输出: [1,3,12,0,0]
# 示例 2:
# 输入: nums = [0]
# 输出: [0]
```

```
# 进阶：你能尽量减少完成的操作次数吗？

def solution4(nums):
    l = 0
    for r in range(len(nums)):
        if nums[r] == 0:
            continue
        nums[l],nums[r]=nums[r],nums[l]
        l += 1

def main():
    nums = list(map(int,input().strip().split()))
    solution4(nums)
    print(nums)

if __name__ == "__main__":
    main()
```

5.

```
# medium
# 5. 盛水最多的容器
# 给定一个长度为 n 的整数数组 height 。有 n 条垂线，第 i 条线的两个端点是 (i, 0) 和 (i, height[i]) 。
# 找出其中的两条线，使得它们与 x 轴共同构成的容器可以容纳最多的水。
# 返回容器可以储存的最大水量。
# 说明：你不能倾斜容器。

# 示例1:
# 输入：[1,8,6,2,5,4,8,3,7]
# 输出：49
# 示例2:
# 输入：height = [1,1]
# 输出：1

def solution5(height):
    water = 0
    l = 0
    r = len(height) - 1
    while l < r:
        if height[l] <= height[r]:
            water = max(water,height[l]*(r-l))
            l += 1
        else:
            water = max(water,height[r]*(r-l))
            r -= 1
    return water

def main():
    height = list(map(int,input().strip().split()))
    water = solution5(height)
    print (water)
```

```
if __name__ == "__main__":
    main()
```

6.

```
# medium
# 6. 三数之和
# 给你一个整数数组 nums ，判断是否存在三元组 [nums[i], nums[j], nums[k]] 满足 i != j、i != k 且 j
!= k ，同时还满足 nums[i] + nums[j] + nums[k] == 0 。请你返回所有和为 0 且不重复的三元组。
# 注意：答案中不可以包含重复的三元组。

# 输入：nums = [-1,0,1,2,-1,-4]
# 输出：[[-1,-1,2],[-1,0,1]]

# 输入：nums = [0,1,1]
# 输出：[]

# 输入：nums = [0,0,0]
# 输出：[[0,0,0]]

def s6(nums:list[int]) -> list[list[int]]:
    res = []
    if not nums or len(nums) < 3:
        return []
    nums.sort()
    for i in range(len(nums)):
        if nums[i] > 0:
            return res
        if nums[i] == nums[i+1]:
            continue
        l = i+1
        r = len(nums)-1
        while l<r:
            sum_lri = nums[i]+nums[l]+nums[r]
            if sum_lri < 0:
                l+=1
            if sum_lri > 0:
                r-=1
            if sum_lri == 0:
                res.append([nums[i],nums[l],nums[r]])
                while l<r and nums[l]==nums[l+1]:
                    l+=1
                while l<r and nums[r]==nums[r-1]:
                    r-=1
                l += 1
                r -= 1
    return res

def main():
    nums = list(map(int,input().strip().split()))
    res = s6(nums)
    print (res)
```

```
if __name__ == "__main__":
    main()
```

7.

```
# medium
#  7. 无重复字符的最长子串
# 给定一个字符串 s ，请你找出其中不含有重复字符的 最长子串的长度。
# 输入: s = "abcabcbb"
# 输出: 3
# 输入: s = "bbbbb"
# 输出: 1
# 输入: s = "pwwkew"
# 输出: 3

def s7(s):
    res = 0
    hash_map = {}
    l = 0
    for r in range(len(s)):
        hash_map[s[r]] = hash_map.get(s[r],0) + 1
        if(len(hash_map)==r-l+1):
            res = max(res,r-l+1)
        while r-l+1>len(hash_map):
            head = s[l]
            hash_map[head] -= 1
            if hash_map[head] == 0:
                del hash_map[head]
            l+=1
    return res

def main():
    s = input().strip()
    res = s7(s)
    print (res)

if __name__ == "__main__":
    main()
```

8.

```
# medium
# 8.找到字符串中所有字母异位词
# 给定两个字符串 s 和 p，找到 s 中所有 p 的 异位词 的子串，返回这些子串的起始索引。不考虑答案输出的顺序。
# 输入: s = "cbaebabacd", p = "abc"
# 输出: [0,6]
# 输入: s = "abab", p = "ab"
# 输出: [0,1,2]

def s8(s,p):
```

```
    res = []
    len_s = len(s)
    len_p = len(p)
    cnt_p = [0] * 26
    cnt_s = [0] * 26
    if len_s < len_p:
        return []
    for i in range(len_p):
        cnt_s[ord(s[i])-ord("a")] += 1
        cnt_p[ord(p[i])-ord("a")] += 1
    if cnt_p == cnt_s:
        res.append(0)
    for i in range(len_p,len_s):
        cnt_s[ord(s[i]) - ord("a")] += 1
        cnt_s[ord(s[i-len_p]) - ord("a")] -= 1
        if cnt_s == cnt_p:
            res.append(i - len_p + 1)
    return res

def main():
    s = input().strip()
    p = input().strip()
    res = s8(s,p)
    print(res)

if __name__ == "__main__":
    main()
```

9.

```
# medium
# 9. 和为K的子数组
# 给你一个整数数组 nums 和一个整数 k ，请你统计并返回 该数组中和为 k 的子数组的个数 。
# 子数组是数组中元素的连续非空序列。
# 输入：nums = [1,1,1], k = 2
# 输出：2
# 输入：nums = [1,2,3], k = 3
# 输出：2

# 思路：算出每一个点的presum，做差找k

def s9(nums,k):
    ans = 0
    len_num = len(nums)
    pre_sum = [0]*(len_num+1)
    pre_sum[0] = 0
    for i in range(1,len_num+1):
        pre_sum[i] = pre_sum[i-1]+nums[i-1]
    for l in range(len_num+1):
        r = l+1
        while r < len_num+1:
            if pre_sum[r]-pre_sum[l] == k:
```

```
                ans += 1
                r += 1
            elif pre_sum[r]-pre_sum[l] < k:
                r += 1
            elif pre_sum[r]-pre_sum[l] > k:
                break
    return ans

nums = list(map(int,input().strip().split()))
k = int(input())
ans=s9(nums,k)
print(ans)
```

10.

```
# medium
# 10. 最大子数和
# 给你一个整数数组 nums ，请你找出一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。
# 子数组是数组中的一个连续部分。
# 输入：nums = [-2,1,-3,4,-1,2,1,-5,4]
# 输出：6
# 输入：nums = [1]
# 输出：1
# 输入：nums = [5,4,-1,7,8]
# 输出：23

def s10(nums):
    len_nums = len(nums)
    if len_nums == 0:
        return 0
    dp = [0]*len_nums
    dp[0] = nums[0]
    for i in range(1,len_nums):
        if dp[i-1]<0:
            dp[i] = nums[i]
        if dp[i-1]>=0:
            dp[i] = nums[i]+dp[i-1]
    return max(dp)

nums=list(map(int,input().strip().split()))
res = s10(nums)
print (res)
```

11.

```
# medium
# 11. 合并区间
# 以数组 intervals 表示若干个区间的集合，其中单个区间为 intervals[i] = [starti, endi] 。请你合并所有
重叠的区间，并返回 一个不重叠的区间数组，该数组需恰好覆盖输入中的所有区间 。
# 输入：intervals = [[1,3],[2,6],[8,10],[15,18]]
# 输出：[[1,6],[8,10],[15,18]]
```

```
# 输入：intervals = [[1,4],[4,5]]
# 输出：[[1,5]]

def s11(intervals):
    intervals.sort(key=lambda p:p[0])
    ans = []
    for p in intervals:
        if ans and p[0] <= ans[-1][1]:
            ans[-1][1] = max(ans[-1][1],p[1])
        else:
            ans.append(p)
    return ans

intervals = []
while True:
    nums = list(map(int,input().strip().split()))
    if nums == []:
        break
    intervals.append(nums)
ans = s11(intervals)
print(ans)
```

12.

```
# medium
# 12.轮转数组
# 给定一个整数数组 nums，将数组中的元素向右轮转 k 个位置，其中 k 是非负数。
# 输入: nums = [1,2,3,4,5,6,7], k = 3
# 输出: [5,6,7,1,2,3,4]
# 输入：nums = [-1,-100,3,99], k = 2
# 输出：[3,99,-1,-100]

nums = list(map(int,input().strip().split()))
k = int(input())
n = k % len(nums)
nums[:]=nums[-n:]+nums[:-n]
print(nums)
```

13.

```
# medium
# 13.除自身以外数组的乘积
# 给你一个整数数组 nums，返回 数组 answer ，其中 answer[i] 等于 nums 中除 nums[i] 之外其余各元素的乘积
。
# 题目数据 保证 数组 nums之中任意元素的全部前缀元素和后缀的乘积都在  32 位 整数范围内。
# 请 不要使用除法，且在 O(n) 时间复杂度内完成此题。
# 输入: nums = [1,2,3,4]
# 输出: [24,12,8,6]
# 输入: nums = [-1,1,0,-3,3]
# 输出: [0,0,9,0,0]
```

```
nums = list(map(int,input().strip().split()))
len_nums = len(nums)
ans = [1]*len_nums
tmp = 1
for i in range(1,len_nums):
    ans[i] = ans[i-1]*nums[i-1]
for i in range(len_nums-2,-1,-1):
    tmp = nums[i+1]*tmp
    ans[i] = tmp*ans[i]
print(ans)
```

14.

```
# medium
# 14.矩阵置零
# 给定一个 m x n 的矩阵，如果一个元素为 0 ，则将其所在行和列的所有元素都设为 0 。请使用 原地 算法。
# 输入：matrix = [[1,1,1],[1,0,1],[1,1,1]]
# 输出：[[1,0,1],[0,0,0],[1,0,1]]
# 输入：matrix = [[0,1,2,0],[3,4,5,2],[1,3,1,5]]
# 输出：[[0,0,0,0],[0,4,5,0],[0,3,1,0]]

m = int(input())
n = int(input())
matrix = list(list())
for i in range(n):
    nums = list(map(int,input().strip().split()))
    matrix.append(nums)
flag_row1_have0 = 0
for item in matrix[0]:
    if item == 0:
        flag_row1_have0 = 1
for row in range(1,len(matrix)):
    flag = 0
    for i in range(len(matrix[row])):
        if matrix[row][i] == 0:
            matrix[0][i] = 0
            flag = 1
    if flag == 1:
        for i in range(len(matrix[row])):
            matrix[row][i] = 0
# print(matrix)
for i in range(len(matrix[0])):
    if matrix[0][i]==0:
        for j in range(len(matrix)):
            matrix[j][i] = 0
# print(matrix)
if flag_row1_have0 == 1:
    for i in range(len(matrix[0])):
        matrix[0][i] = 0

print(matrix)
```

15.

```
# medium
# 15. 螺旋矩阵
# 给你一个 m 行 n 列的矩阵 matrix ，请按照 顺时针螺旋顺序 ，返回矩阵中的所有元素。
# 输入：matrix = [[1,2,3],[4,5,6],[7,8,9]]
# 输出：[1,2,3,6,9,8,7,4,5]
# 输入：matrix = [[1,2,3,4],[5,6,7,8],[9,10,11,12]]
# 输出：[1,2,3,4,8,12,11,10,9,5,6,7]

def spiralOrder(matrix):
        left = 0
        right = len(matrix[0])-1
        up = 0
        down = len(matrix)-1
        sumnum = (down+1) * (right+1)
        # print(sumnum)
        direction = 1
        res = []
        while len(res) < sumnum:
            if direction == 1:
                for i in range(left, right + 1):
                    res.append(matrix[up][i])
                up += 1
                direction = 2
            elif direction == 2:
                for i in range(up, down + 1):
                    res.append(matrix[i][right])
                right -= 1
                direction = 3
            elif direction == 3:
                for i in range(right, left - 1, -1):
                    res.append(matrix[down][i])
                down -= 1
                direction = 4
            elif direction == 4:
                for i in range(down, up - 1, -1):
                    res.append(matrix[i][left])
                left += 1
                direction = 1
        return res

def main():
    matrix = list(list())
    m = int(input())
    for i in range(m):
        nums = list(map(int,input().strip().split()))
        matrix.append(nums)
    res=spiralOrder(matrix)
    print (res)
```

```
if __name__ =="__main__":
    main()
```

16.

```
# medium
# 16.旋转图像
# 给定一个 n × n 的二维矩阵 matrix 表示一个图像。请你将图像顺时针旋转 90 度。
# 你必须在 原地 旋转图像，这意味着你需要直接修改输入的二维矩阵。请不要 使用另一个矩阵来旋转图像。

# 输入：matrix = [[1,2,3],[4,5,6],[7,8,9]]
# 输出：[[7,4,1],[8,5,2],[9,6,3]]
# 输入：matrix = [[5,1,9,11],[2,4,8,10],[13,3,6,7],[15,14,12,16]]
# 输出：[[15,13,2,5],[14,3,4,1],[12,6,8,9],[16,7,10,11]]

def solution(matrix):
    n = len(matrix)
    for i in range(n // 2):
        for j in range((n + 1) // 2):
            tmp = matrix[i][j]
            matrix[i][j] = matrix[n - 1 - j][i]
            matrix[n - 1 - j][i] = matrix[n - 1 - i][n - 1 - j]
            matrix[n - 1 - i][n - 1 - j] = matrix[j][n - 1 - i]
            matrix[j][n - 1 - i] = tmp
            # print(matrix)

m = int(input())
matrix = list(list())
for i in range(m):
    nums = list(map(int,input().strip().split()))
    matrix.append(nums)
solution(matrix)
print(matrix)

# while True:
#     try:
#         line = input().strip()
#         if not line:
#             break
#         nums = list(map(int, line.split()))
#         matrix.append(nums)
```

17.

```
# medium
# 搜索二维矩阵
# 编写一个高效的算法来搜索 m x n 矩阵 matrix 中的一个目标值 target 。该矩阵具有以下特性：
# 每行的元素从左到右升序排列。
# 每列的元素从上到下升序排列。
```

```
# 输入：matrix = [[1,4,7,11,15],[2,5,8,12,19],[3,6,9,16,22],[10,13,14,17,24],
[18,21,23,26,30]], target = 5
# 输出：true

# 输入：matrix = [[1,4,7,11,15],[2,5,8,12,19],[3,6,9,16,22],[10,13,14,17,24],
[18,21,23,26,30]], target = 20
# 输出：false

def searchMatrix(matrix: list[list[int]], target: int) -> bool:
    i = len(matrix) - 1
    j = 0
    while j < len(matrix[0]) and i >= 0:
        # print([i,j])
        if matrix[i][j] == target:
            return True
        elif matrix[i][j] > target:
            i -= 1
        elif matrix[i][j] < target:
            j += 1
    return False

m = int(input())
matrix = list(list())
for i in range(m):
    nums = list(map(int,input().strip().split()))
    matrix.append(nums)
target = int(input())
print(searchMatrix(matrix,target))
```

18.

```
# medium
# 18.环形链表
# 给定一个链表的头节点  head ，返回链表开始入环的第一个节点。 如果链表无环，则返回 null。
# 如果链表中有某个节点，可以通过连续跟踪 next 指针再次到达，则链表中存在环。 为了表示给定链表中的环，评测系统内部使用整数 pos 来表示链表尾连接到链表中的位置（索引从 0 开始）。如果 pos 是 -1，则在该链表中没有环。注意：pos 不作为参数进行传递，仅仅是为了标识链表的实际情况。
# 不允许修改 链表。

# 输入：head = [3,2,0,-4], pos = 1
# 输出：返回索引为 1 的链表节点

# 输入：head = [1,2], pos = 0
# 输出：返回索引为 0 的链表节点

# 输入：head = [1], pos = -1
# 输出：返回 null

class Solution:
    def detectCycle(self, head):
        slow, fast = head, head
```

```
        while True:
            if not fast or not fast.next:
                return None
            slow = slow.next
            fast = fast.next.next
            if slow == fast:
                break;
        fast = head
        while slow != fast:
            fast = fast.next
            slow = slow.next
        return slow
```

19.

```
# medium
# 19. 从前序和中序遍历序列构造二叉树
# 给定两个整数数组 preorder 和 inorder ，其中 preorder 是二叉树的先序遍历， inorder 是同一棵树的中序遍历，请构造二叉树并返回其根节点。
# 输入: preorder = [3,9,20,15,7], inorder = [9,3,15,20,7]
# 输出: [3,9,20,null,null,15,7]
# 输入: preorder = [-1], inorder = [-1]
# 输出: [-1]

# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution(object):
    def buildTree(self, preorder, inorder):
        if not preorder or not inorder:
            return None

        # 前序遍历的第一个节点就是根节点
        root = TreeNode(preorder[0])

        # 在中序遍历中找到根节点的位置
        mid = inorder.index(preorder[0])

        # 递归构建左子树和右子树
        # 左子树：前序遍历[1:mid+1]，中序遍历[:mid]
        root.left = self.buildTree(preorder[1:mid+1], inorder[:mid])
        # 右子树：前序遍历[mid+1:]，中序遍历[mid+1:]
        root.right = self.buildTree(preorder[mid+1:], inorder[mid+1:])

        return root
```

20.

```
# medium
# 20.岛屿数量
# 给你一个由 '1'（陆地）和 '0'（水）组成的的二维网格，请你计算网格中岛屿的数量。
# 岛屿总是被水包围，并且每座岛屿只能由水平方向和/或竖直方向上相邻的陆地连接形成。
# 此外，你可以假设该网格的四条边均被水包围。

#  输入：grid = [
#   ["1","1","1","1","0"],
#   ["1","1","0","1","0"],
#   ["1","1","0","0","0"],
#   ["0","0","0","0","0"]
# ]
# 输出：1

# 输入：grid = [
#   ["1","1","0","0","0"],
#   ["1","1","0","0","0"],
#   ["0","0","1","0","0"],
#   ["0","0","0","1","1"]
# ]
# 输出：3

def solution(grid):
    def dfs(grid,i,j):
        if not 0<=i<len(grid) or not 0<=j<len(grid[0]) or grid[i][j]==0:
            return
        grid[i][j] = 0
        dfs(grid, i + 1, j)
        dfs(grid, i, j + 1)
        dfs(grid, i - 1, j)
        dfs(grid, i, j - 1)
    res = 0
    for i in range(len(grid)):
        for j in range(len(grid[0])):
            if grid[i][j] == 1:
                dfs(grid,i,j)
                res += 1
    return res

grid = list(list())
m = int(input())
for i in range(m):
    nums = list(map(int,input().strip().split()))
    grid.append(nums)
print(solution(grid))
```

21.

```python
# medium
# 21.腐烂的橘子
# 在给定的 m x n 网格 grid 中，每个单元格可以有以下三个值之一：
# 值 0 代表空单元格；
# 值 1 代表新鲜橘子；
# 值 2 代表腐烂的橘子。
# 每分钟，腐烂的橘子 周围 4 个方向上相邻 的新鲜橘子都会腐烂。
# 返回 直到单元格中没有新鲜橘子为止所必须经过的最小分钟数。如果不可能，返回 -1 。

# 输入：grid = [[2,1,1],[1,1,0],[0,1,1]]
# 输出：4
# 输入：grid = [[2,1,1],[0,1,1],[1,0,1]]
# 输出：-1
# 输入：grid = [[0,2]]
# 输出：0

from collections import deque

def orangesRotting(grid):
    if not grid or not grid[0]:
        return 0

    rows, cols = len(grid), len(grid[0])
    queue = deque()
    fresh_count = 0

    # 找到所有腐烂橘子的位置，统计新鲜橘子数量
    for i in range(rows):
        for j in range(cols):
            if grid[i][j] == 2:
                queue.append((i, j))
            elif grid[i][j] == 1:
                fresh_count += 1

    # 如果没有新鲜橘子，直接返回0
    if fresh_count == 0:
        return 0

    # 四个方向：上、下、左、右
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    minutes = 0

    # BFS模拟腐烂过程
    while queue:
        size = len(queue)
        has_rotted = False

        # 处理当前这一轮的所有腐烂橘子
        for _ in range(size):
            x, y = queue.popleft()
```

```
            # 向四个方向传播
            for dx, dy in directions:
                nx, ny = x + dx, y + dy

                # 检查边界和是否为新鲜橘子
                if 0 <= nx < rows and 0 <= ny < cols and grid[nx][ny] == 1:
                    grid[nx][ny] = 2  # 变腐烂
                    queue.append((nx, ny))
                    fresh_count -= 1
                    has_rotted = True

        # 如果这一轮有橘子腐烂，时间+1
        if has_rotted:
            minutes += 1

    # 如果还有新鲜橘子剩余，返回-1
    return minutes if fresh_count == 0 else -1

m = int(input())
grid = list(list())
for i in range(m):
    nums = list(map(int,input().strip().split()))
    grid.append(nums)
print(orangesRotting(grid))
```

22.

```
# medium
# 22.括号生成
# 数字 n 代表生成括号的对数，请你设计一个函数，用于能够生成所有可能的并且 有效的 括号组合。
# 输入：n = 3
# 输出：["((()))","(()())","(())()","()(())","()()()"]
# 输入：n = 1
# 输出：["()"]

n = int(input())
ans = []
def solu(S,l,r):
    if len(S)==2*n:
        ans.append("".join(S))
        return
    if l<n:
        S.append('(')
        solu(S,l+1,r)
        S.pop()
    if r<l:
        S.append(')')
        solu(S,l,r+1)
        S.pop()
solu([],0,0)
```

```
print(ans)
print(len(ans))
```

23.

```
# medium
# 23. 排序数组找元素第一个和最后一个位置
# 给你一个按照非递减顺序排列的整数数组 nums，和一个目标值 target。请你找出给定目标值在数组中的开始位置和结束位置。
# 如果数组中不存在目标值 target，返回 [-1, -1]。
# 你必须设计并实现时间复杂度为 O(log n) 的算法解决此问题。

nums = list(map(int,input().strip().split()))
target = int(input())

def searchLeft():
    l,r = 0, len(nums)-1
    while l<=r:
        mid = (r-l)//2+l
        if nums[mid] == target:
            if mid==0 or nums[mid-1]<target:
                return mid
            else:
                r = mid-1
        elif nums[mid] < target:
            l = mid+1
        else :
            r =mid-1
    return -1

def searchRight():
    l,r = 0, len(nums)-1
    while l<=r:
        mid = (r-l)//2+l
        if nums[mid] == target:
            if mid==len(nums)-1 or nums[mid+1]>target:
                return mid
            else:
                r = mid-1
        elif nums[mid] < target:
            l = mid+1
        else :
            r =mid-1
    return -1

print([searchLeft(),searchRight()])
```

24.

```
# medium
# 24.每日温度
# 给定一个整数数组 temperatures ，表示每天的温度，返回一个数组 answer ，其中 answer[i] 是指对于第 i 天，下一个更高温度出现在几天后。如果气温在这之后都不会升高，请在该位置用 0 来代替。
# 输入: temperatures = [73,74,75,71,69,72,76,73]
# 输出: [1,1,4,2,1,1,0,0]
# 输入: temperatures = [30,40,50,60]
# 输出: [1,1,1,0]
# 输入: temperatures = [30,60,90]
# 输出: [1,1,0]

nums = list(map(int, input().strip().split()))

def daily_temperatures(temperatures):
    n = len(temperatures)
    result = [0] * n  # 初始化结果数组，默认都是0
    stack = []  # 单调栈，存储索引

    for i in range(n):
        # 当栈不为空且当前温度大于栈顶索引对应的温度时
        while stack and temperatures[i] > temperatures[stack[-1]]:
            prev_index = stack.pop()  # 弹出栈顶索引
            result[prev_index] = i - prev_index  # 计算距离

        stack.append(i)  # 将当前索引入栈

    return result

# 调用函数并输出结果
answer = daily_temperatures(nums)
print(answer)
```

25.

```
# 往年题
# 统计次数
# **时间限制：** 1.0 秒
# **空间限制：** 512 MB
# ## 题目描述
# 给定两个正整数 $n$ 和 $k~(1 \le k \le 9)$，求从 1 到 $n$ 这 $n$ 个正整数的十进制表示中 $k$ 出现的次数。
# ## 输入格式
# 从标准输入读入数据。
# 输入的第一行包含两个正整数 $n$ 和 $k$，保证 $n \le 10^6$ 和 $1 \le k \le 9$ 。
# ## 输出格式
# 输出到标准输出。
# 输出一个整数，表示答案。
# ## 样例 1 输入
# 12 1
```

```
# ## 样例 1 输出
# 5
# ## 样例 1 解释
# 从 $1$ 到 $12$ 这些整数中包含 $1$ 的数字有 $1,10,11,12$，一共出现了 $5$ 次 $1$ 。

# import time
# start_time = time.perf_counter()

# n, k = map(int, input().split())
# count = 0
# k_str = str(k)

# for i in range(1, n + 1):
#     count += str(i).count(k_str)

# print(count)

# end_time = time.perf_counter()
# print(f"执行时间: {end_time - start_time:.4f} 秒")

import time

def count_digit_fast(n, k):
    if n <= 0:
        return 0

    s = str(n)
    length = len(s)
    count = 0

    for i in range(length):
        # 当前位左边的数字
        left = int(s[:i]) if i > 0 else 0
        # 当前位的数字
        cur = int(s[i])
        # 当前位右边的数字个数
        right_count = length - i - 1
        # 当前位的权重
        power = 10 ** right_count

        if cur < k:
            count += left * power
        elif cur == k:
            right = int(s[i+1:]) if i < length - 1 else 0
            count += left * power + right + 1
        else:  # cur > k
            count += (left + 1) * power

    return count

start_time = time.perf_counter()

n, k = map(int, input().split())
```

```
result = count_digit_fast(n, k)
print(result)

end_time = time.perf_counter()
print(f"执行时间: {end_time - start_time:.6f} 秒")
```

26.

```
# # 等差数列
# **时间限制：** 1.0 秒
# **空间限制：** 512 MB
# ## 题目描述
# 有一个特殊的 $n$ 行 $m$ 列的矩阵 $A_{ij}~(1 \le i \le n,~1 \le j \le m)$，每个元素都是正整数，每一行和每一列都是独立的等差数列。在某一次故障中，这个矩阵的某些元素的真实值丢失了，被重置为 $0$。现在需要你想办法恢复这些元素，并且按照行号和列号从小到大的顺序（行号为第一关键字，列号为第二关键字，从小到大）输出能够恢复的元素。
# ## 输入格式
# 从标准的输入读入数据。
# 输入的第一行包含两个正整数 $n$ 和 $m$，保证 $n \le 10^3$ 和 $m \le 10^3$ 。
# 接下来 $n$ 行，每行 $m$ 个整数，表示整个矩阵，保证 $1 \le A_{ij} \le 10^9$。如果 $A_{ij}$ 等于 $0$，表示真实值丢失的元素。
# ## 输出格式
# 输出到标准输出。
# 输出若干行，表示所有能够恢复的元素。每行三个整数 $i,j,x$，表示 $A_{ij}$ 的真实值是 $x$ 。
# ## 样例 1 输入
# 3 4
# 1 2 0 0
# 0 0 0 0
# 3 0 0 0
# ## 样例 1 输出
# 1 3 3
# 1 4 4
# 2 1 2
# ## 样例 1 解释
# 可以恢复 $3$ 个元素，$A_{13}$ 的真实值是 $3$，$A_{14}$ 的真实值是 $4$，$A_{21}$ 的真实值是 $2$ 。

# 读取输入
n, m = map(int, input().split())
matrix = []
original_matrix = []  # 保存原始矩阵用于判断哪些是恢复的
for i in range(n):
    nums = list(map(int, input().strip().split()))
    matrix.append(nums[:])  # 工作矩阵
    original_matrix.append(nums[:])  # 原始矩阵

# 存储每行每列的等差数列信息
hang_info = {}  # 行信息: {行号: (首项, 公差)}
lie_info = {}   # 列信息: {列号: (首项, 公差)}

def get_arithmetic_info(arr):
    """从数组中获取等差数列信息，返回(首项, 公差)或None"""
    non_zero = [(i, val) for i, val in enumerate(arr) if val != 0]
```

```
    if len(non_zero) < 2:
        return None

    # 用前两个非零元素计算公差
    pos1, val1 = non_zero[0]
    pos2, val2 = non_zero[1]

    if pos2 == pos1:
        return None

    d = (val2 - val1) // (pos2 - pos1)
    a1 = val1 - d * pos1

    # 验证所有非零元素是否符合等差数列
    for pos, val in non_zero:
        if a1 + d * pos != val:
            return None

    if a1 <= 0:  # 确保首项为正
        return None

    return (a1, d)

# 反复尝试恢复，直到无法继续
changed = True
while changed:
    changed = False

    # 尝试确定每行的等差数列信息
    for i in range(n):
        if i not in hang_info:
            info = get_arithmetic_info(matrix[i])
            if info:
                hang_info[i] = info
                # 用等差数列信息填充该行的0
                a1, d = info
                for j in range(m):
                    if matrix[i][j] == 0:
                        matrix[i][j] = a1 + d * j
                        changed = True

    # 尝试确定每列的等差数列信息
    for j in range(m):
        if j not in lie_info:
            col = [matrix[i][j] for i in range(n)]
            info = get_arithmetic_info(col)
            if info:
                lie_info[j] = info
                # 用等差数列信息填充该列的0
                a1, d = info
                for i in range(n):
                    if matrix[i][j] == 0:
```

```
                        matrix[i][j] = a1 + d * i
                        changed = True

# 收集并输出恢复的元素
results = []
for i in range(n):
    for j in range(m):
        if original_matrix[i][j] == 0 and matrix[i][j] != 0:
            results.append((i+1, j+1, matrix[i][j]))

# 按行号和列号排序输出
results.sort()
for i, j, x in results:
    print(i, j, x)
```

27.

```
# # Prime
# **时间限制：** 0.2 秒
# **空间限制：** 512 MB
# ## 题目描述
# 输入一个正整数 $x$，请你在 $x$ 后面添加若干位数字（不能不添加；添加的部分不得以数字 0 开头），使得结果为质数，在这个前提下所得的结果应尽量小。
# ## 输入格式
# 从标准输入读入数据。
# 输入一行，输入一个正整数 $x$ 。
# ## 输出格式
# 输出到标准输出。
# 输出一行，包含一个整数，表示所得的结果。
# 输入保证 $1 \le x \le 100$ 。
# 本题共有 100 个测试点，每个测试点 1 分。

def isPrime(num):
    if num < 2:
        return False
    if num == 2:
        return True
    if num % 2 == 0:
        return False
    import math
    for i in range(3, int(math.sqrt(num)) + 1, 2):
        if num % i == 0:
            return False
    return True

def get_result(num):
    num = str(num)
    n = 1
    while True:
        tmp = int(num+str(n))
        if isPrime(tmp):
            return tmp
```

```
        else:
            n += 1

def get_100list():
    list_100 = list()
    for i in range(1,101):
        list_100.append(get_result(i))
    print(list_100)
    return list_100

list_100 = [11, 23, 31, 41, 53, 61, 71, 83, 97, 101, 113, 127, 131, 149, 151, 163, 173, 181,
191, 2011, 211, 223, 233, 241, 251, 263, 271, 281, 293, 307, 311, 3217, 331, 347, 353, 367,
373, 383, 397, 401, 419, 421, 431, 443, 457, 461, 479, 487, 491, 503, 5113, 521, 5323, 541,
557, 563, 571, 587, 593, 601, 613, 6211, 631, 641, 653, 661, 673, 683, 691, 701, 719, 727,
733, 743, 751, 761, 773, 787, 797, 809, 811, 821, 839, 8419, 853, 863, 877, 881, 8923, 907,
911, 929, 937, 941, 953, 967, 971, 983, 991, 1009]
n = int(input())
print(list_100[n-1])
```

28.

```
# # Friend
# **时间限制：** 1.0 秒
# **空间限制：** 512 MB
# ## 题目描述
# F 学校有 $n$ 个学生，编号为 $1, 2, \cdots, n$。这些学生之间存在 $m$ 对好友关系。每对好友关系形如：$u_j$ 号学生与 $v_j$ 号学生互为好友 $(1 \le j \le m)$。好友关系是双向的，这意味着 $u_j$ 号学生是 $v_j$ 号学生的好友，同时 $v_j$ 号学生也是 $u_j$ 号学生的好友。
# F 学校要将 $n$ 个学生均匀（等概率）随机地分为若干小组，每组 3 个学生。保证 $n$ 是 3 的倍数，即能够恰好分完。在分组完毕后，每个组内的好友关系也会有不同的情况。现在，对于每个学生 $i$，他希望计算他所在小组的 3 个学生当中以下每个事件发生的概率：
# 1. 3 个学生两两均不为好友；
# 1. 3 个学生中，除自己外的 2 个学生互为好友，不存在其他好友关系；
# 1. 3 个学生中，自己与另外某个学生互为好友，不存在其他好友关系；
# 1. 3 个学生中，恰好有 2 对好友关系，且有 2 个好友的那个人是自己（即：自己与另外 2 个学生分别互为好友，但他们两个不为好友）；
# 1. 3 个学生中，恰好有 2 对好友关系，但有 2 个好友的那个人不是自己（即：存在某个学生 A 与自己和另外一个学生 B（分别）互为好友，但自己与 B 不为好友）；
# 1. 3 个学生中两两互为好友。
# 请帮助每个学生计算吧！

# ## 输入格式
# 从标准输入读入数据。
# 第一行输入两个正整数 $n, m$，以空格隔开。
# 接下来 $m$ 行，每行输入两个正整数 $u_j, v_j(u_j\ne v_j)$，以空格隔开，表示 $u_j$ 与 $v_j$ 号学生互为好友。
# 数据保证不存在两组重复的好友关系。
# ## 输出格式
# 输出到标准输出。
# 输出 $n$ 行，每行 6 个最简分数，以空格隔开，表示每个学生每种情况的发生概率。
# 输出最简分数的形式为：先输出分子，再输出斜线 $/$，最后输出分母。你应当输出最简分数，例如不应当输出 $3/6$，而应输出 $1/2$ 。
```

```python
# 特殊地，如果所求的某个概率为 $0$，应当输出 $0/1$；概率为 $1$ 则输出 $1/1$ 。
# ## 样例 1 输入
# 3 2
# 1 2
# 1 3
# ## 样例 1 输出
# 0/1 0/1 0/1 1/1 0/1 0/1
# 0/1 0/1 0/1 0/1 1/1 0/1
# 0/1 0/1 0/1 0/1 1/1 0/1
# ## 样例 1 解释
# 一共只有 3 个学生，分组实际上仅有一种方案，不存在随机性。
# 3 个学生之间存在 2 对好友关系，在 1 号学生看来是第 4 种情况，在 2 号和 3 号学生看来是第 5 种情况。
# ## 样例 2 输入
# 6 6
# 1 2
# 2 3
# 3 1
# 1 4
# 1 5
# 1 6
# ## 样例 2 输出
# 0/1 0/1 0/1 9/10 0/1 1/10
# 3/10 0/1 3/10 0/1 3/10 1/10
# 3/10 0/1 3/10 0/1 3/10 1/10
# 1/2 1/10 0/1 0/1 2/5 0/1
# 1/2 1/10 0/1 0/1 2/5 0/1
# 1/2 1/10 0/1 0/1 2/5 0/1

from fractions import Fraction
from itertools import combinations
import math

def solve():
    n, m = map(int, input().split())

    # 构建好友关系图
    friends = [set() for _ in range(n + 1)]
    for _ in range(m):
        u, v = map(int, input().split())
        friends[u].add(v)
        friends[v].add(u)

    # 生成所有可能的3人组合
    all_students = list(range(1, n + 1))
    all_groups = list(combinations(all_students, 3))

    # 计算总的分组方案数
    # 这是一个复杂的组合数学问题，需要计算将n个人分成n/3个3人组的方案数
    total_ways = calculate_total_grouping_ways(n)

    # 为每个学生初始化6种情况的计数
    student_counts = [[0] * 6 for _ in range(n + 1)]
```

```
    # 为每个学生计算他参与的所有可能3人组
    for student in range(1, n + 1):
        # 找到包含该学生的所有3人组
        student_groups = [group for group in all_groups if student in group]

        # 计算该学生被分到每个组的概率权重
        for group in student_groups:
            a, b, c = group

            # 计算这个特定3人组在所有分组方案中出现的次数
            group_weight = calculate_group_weight(group, n)

            # 判断这个组属于哪种情况（从学生的视角）
            case = classify_group(student, group, friends)

            # 累加到对应情况的计数中
            student_counts[student][case] += group_weight

    # 输出结果
    for student in range(1, n + 1):
        results = []
        for case in range(6):
            prob = Fraction(student_counts[student][case], total_ways)
            results.append(f"{prob.numerator}/{prob.denominator}")
        print(" ".join(results))

def calculate_total_grouping_ways(n):
    """计算将n个人分成n/3个3人组的总方案数"""
    groups = n // 3
    # 公式：n! / (3!^(n/3) * (n/3)!)
    numerator = math.factorial(n)
    denominator = (math.factorial(3) ** groups) * math.factorial(groups)
    return numerator // denominator

def calculate_group_weight(group, n):
    """计算特定3人组在所有分组方案中出现的次数"""
    remaining = n - 3
    if remaining == 0:
        return 1
    # 剩余remaining个人分成remaining/3个3人组的方案数
    remaining_groups = remaining // 3
    numerator = math.factorial(remaining)
    denominator = (math.factorial(3) ** remaining_groups) * math.factorial(remaining_groups)
    return numerator // denominator

def classify_group(student, group, friends):
    """判断3人组从指定学生视角属于哪种情况"""
    a, b, c = group

    # 重新排列，让student在第一位
    if student == a:
        me, other1, other2 = a, b, c
    elif student == b:
```

```
        me, other1, other2 = b, a, c
    else:  # student == c
        me, other1, other2 = c, a, b

    # 统计好友关系
    me_other1 = other1 in friends[me]
    me_other2 = other2 in friends[me]
    other1_other2 = other2 in friends[other1]

    total_friendships = sum([me_other1, me_other2, other1_other2])

    if total_friendships == 0:
        return 0  # 情况1：无好友关系
    elif total_friendships == 1:
        if other1_other2:
            return 1  # 情况2：除自己外的2人是好友
        else:
            return 2  # 情况3：自己与某人是好友
    elif total_friendships == 2:
        if me_other1 and me_other2:
            return 3  # 情况4：自己有2个好友
        else:
            return 4  # 情况5：别人有2个好友
    else:  # total_friendships == 3
        return 5  # 情况6：两两都是好友

# 简化版本（适用于小规模数据）
def solve_simple():
    """适用于n较小的情况，直接枚举所有分组方案"""
    n, m = map(int, input().split())

    friends = [set() for _ in range(n + 1)]
    for _ in range(m):
        u, v = map(int, input().split())
        friends[u].add(v)
        friends[v].add(u)

    students = list(range(1, n + 1))
    all_groupings = generate_all_groupings(students)

    # 为每个学生统计6种情况的出现次数
    counts = [[0] * 6 for _ in range(n + 1)]

    for grouping in all_groupings:
        for group in grouping:
            for student in group:
                case = classify_group(student, group, friends)
                counts[student][case] += 1

    total_groupings = len(all_groupings)

    for student in range(1, n + 1):
        results = []
```

```
        for case in range(6):
            prob = Fraction(counts[student][case], total_groupings)
            results.append(f"{prob.numerator}/{prob.denominator}")
        print(" ".join(results))

def generate_all_groupings(students):
    """生成所有可能的分组方案（递归）"""
    if len(students) == 0:
        return [[]]
    if len(students) == 3:
        return [[tuple(students)]]

    result = []
    first = students[0]
    remaining = students[1:]

    # 选择first的两个伙伴
    for i in range(len(remaining)):
        for j in range(i + 1, len(remaining)):
            partner1, partner2 = remaining[i], remaining[j]
            current_group = (first, partner1, partner2)

            # 剩余的学生
            rest = [s for s in remaining if s != partner1 and s != partner2]

            # 递归处理剩余学生
            sub_groupings = generate_all_groupings(rest)

            for sub_grouping in sub_groupings:
                result.append([current_group] + sub_grouping)

    return result

if __name__ == "__main__":
    solve_simple()  # 对于比赛，可能需要根据数据规模选择不同的实现
```

29.

```
# # 公司
# **时间限制：** 1.0 秒
# **空间限制：** 512 MB
# ## 题目描述
# 给定一个有 $n$ 个雇员的初创公司，雇员从 $1$ 到 $n$ 编号，编号为 $i$ 的人有一个固定的薪资 $a_i$ 。最初所有人都不知道公司里其他员工的薪资。
# 某一天由于公司数据库发生问题，泄露了 $m$ 条数据，导致有一部分人知道了其他部分人的薪资。其中对于编号为 $i$ 的雇员，设他所了解到的人的平均薪资为 $v_i$ （如果有多条重复的数据，那么也会被计算多次），如果 $a_i<v_i$ 那么他就会萌生想要离职的想法。
# 当然如果一个人不了解其他人的薪资，那么他也不会萌生想要离职的想法。
# 给定所有 $n$ 个人的薪资 $a_i$ ，以及 $m$ 个数对 $(x_i,y_i)$ 表示编号为 $x_i$ 的雇员知道了编号为 $y_i$ 的雇员的薪资，问会有多少雇员萌生离职的想法。
# ## 输入格式
# 从标准输入读入数据。
```

```
# 输入的第一行包含两个正整数 $n,m$ ，分别表示公司的人数和泄露的数据条数。
# 输入的第二行包含 $n$ 个正整数 $a_i$ ，依次表示 $n$ 个人的薪资。
# 接下来 $m$ 行，每行包含两个正整数 $(x_i,y_i)$ 表示编号为 $x_i$ 的雇员知道了编号为 $y_i$ 雇员的薪资。
# ## 输出格式
# 输出到标准输出。
# 输出一个正整数表示对应的答案。
# ## 样例 1 输入
# 4 4
# 10 20 30 40
# 3 2
# 3 4
# 3 4
# 1 2
# ## 样例 1 输出
# 2
# ## 样例 1 解释
# 编号为 $1$ 和 $3$ 的雇员都会萌生离职的想法。
from collections import defaultdict

n,m = map(int, input().split())
salary = list(map(int,input().strip().split()))
secret = defaultdict(list)
result = 0

for i in range(m):
    k,l = map(int,input().split())
    secret[k].append(salary[l-1])

for i in range(n):
    if secret[i+1]:
        sum = 0
        num = 0
        for item in secret[i+1]:
            sum += item
            num += 1
        avr_salary = sum/num
        if avr_salary > salary[i]:
            result += 1

print(result)
```

30.

```
# # 任务调度

# **时间限制：** 3.0 秒

# **空间限制：** 512 MB

# ## 题目描述
```

任务调度是计算机系统中一项重要的工作。今天你的任务，就是模拟一个计算机系统模型的任务调度过程，并给出相应操作的执行结果。

在这个模型中，不同任务按照一定顺序到来，等待被执行。任务处理机制需要维护任务的等待情况，并在相应的时机选择相应的任务进行执行。

不同的任务之间以编号进行区分，为方便起见，按照任务到来的顺序，由先到后编号为 $1,2,3,...$ 。每个任务都拥有一个重要程度 a_i ，所有任务的重要程度两两不同。

在一般情况下，处理任务应当按照任务到来的先后顺序依次处理，也就是说任务等待应当形成一个队列。但考虑到不同任务的重要程度不同，这一原则可能被打破。具体而言，有如下几种操作：

- $1\text{ }a_i$ ：一个新的任务到来，其编号为先前出现过的最大任务编号 $+1$ ，其重要程度为 a_i，在任务等待队列中被安排至队列末尾。考虑到计算机内存限制，同一时刻正在等待的任务数量不能超过 m ，因此如果当前已经有 m 个任务在等待，则这一操作将出现错误。
- $2\text{ }a_i\text{ }x_i$ ：一个新的任务到来，其编号为先前出现过的最大任务编号 $+1$ ，其重要程度为 a_i，在任务等待队列中被安排至任务编号为 x_i 的任务前面并紧挨任务 x_i 的位置。如果当前已有 m 个任务在等待，或任务 x_i 当前不在等待队列中，这一操作将出现错误。
- 3 ：任务处理机制将处理当前排在等待队列队首的任务，并将其从等待队列中移除。若当前等待队列为空，这一操作将出现错误。
- 4 ：任务处理机制将处理当前等待队列中重要程度最大的任务，并将其从等待队列中移除。若当前等待队列为空，这一操作将出现错误。

除上述提到的错误情况外，操作均可以成功执行。

最开始，任务等待队列为空，接下来你需要处理 n 个操作，每个操作形如上述几种之一。对于每个操作，你需要正确判断是否会出现错误，如果出现错误，需要输出一个 `ERR` ，并不予以执行（但对于操作 1 和 2 而言，仍会占用一个新的任务编号）；如果可以成功执行，则需要输出一个正整数，表示这次操作涉及到的任务编号，在操作 1 和 2 中表示新到来的任务编号，操作 3 和 4 中表示被处理的任务编号。

输入格式

从标准输入读入数据。

输入的第一行包含两个正整数 n, m ，分别表示需要执行的操作个数和队伍的最大容量。

接下来 n 行，每行按上述格式描述一个操作。

输出格式

输出到标准输出。

输出 n 行，每行表示对应操作执行的结果，格式如上所述。

样例 1 输入

```
12 3
1 2
1 6
2 1 2
2 7 3
1 5
```
```

```
# 3
# 3
# 1 8
# 2 4 3
# 4
# 4
# 4
# ```

# ## 样例 1 输出

# ```
# 1
# 2
# 3
# ERR
# ERR
# 1
# 3
# 6
# ERR
# 6
# 2
# ERR
# ```

# ## 样例 1 解释

# 第 $4, 5$ 次操作均因等待队列已满而出现错误，第 $9$ 次操作因 $x_i$ 不存在于等待队列中而出现错误，第 $12$ 次操作因等待队列为空而出现错误。

from collections import deque

def solve():
    n, m = map(int, input().split())

    # 任务等待队列，存储任务编号
    queue = deque()
    # 任务重要程度字典
    importance = {}
    # 当前最大任务编号
    max_task_id = 0

    for _ in range(n):
        operation = list(map(int, input().split()))

        if operation[0] == 1:  # 操作1：队尾添加任务
            a_i = operation[1]

            if len(queue) >= m:  # 队列已满
                max_task_id += 1  # 仍要占用编号
                print("ERR")
            else:
```

```
                max_task_id += 1
                queue.append(max_task_id)
                importance[max_task_id] = a_i
                print(max_task_id)

        elif operation[0] == 2:  # 操作2：在指定任务前插入
            a_i = operation[1]
            x_i = operation[2]

            if len(queue) >= m or x_i not in queue:  # 队列已满或目标任务不存在
                max_task_id += 1  # 仍要占用编号
                print("ERR")
            else:
                max_task_id += 1
                # 找到x_i的位置，在其前面插入
                queue_list = list(queue)
                insert_pos = queue_list.index(x_i)
                queue_list.insert(insert_pos, max_task_id)
                queue = deque(queue_list)
                importance[max_task_id] = a_i
                print(max_task_id)

        elif operation[0] == 3:  # 操作3：处理队首任务
            if len(queue) == 0:  # 队列为空
                print("ERR")
            else:
                task_id = queue.popleft()
                del importance[task_id]
                print(task_id)

        elif operation[0] == 4:  # 操作4：处理重要程度最高的任务
            if len(queue) == 0:  # 队列为空
                print("ERR")
            else:
                # 找到重要程度最高的任务
                max_importance = -1
                max_task = -1
                for task_id in queue:
                    if importance[task_id] > max_importance:
                        max_importance = importance[task_id]
                        max_task = task_id

                # 从队列中移除该任务
                queue.remove(max_task)
                del importance[max_task]
                print(max_task)

solve()
```